INTERDATA

FORTRAN V LEVEL II COMPILER

PROJECT NO. 03561

FUNCTIONAL SPECIFICATION

J. BURDA
J. McGOVNEY

January 13, 1975

NOTE
All information contained in this document is subject to change without notice.

1. PROGRAM NAME: FORTRAN V LEVEL II COMPILER

2. PART NUMBER:

3. GENERAL DESCRIPTION:

The FORTRAN V LEVEL II compiler is the FORTRAN software package that is intended to provide Interdata users with FORTRAN capability for years to come. At the time of this writing it is intended to provide them with the American National Standards Institute's latest preliminary FORTRAN standard (X3J3/56) and also to be nearly upwards compatible with Interdata FORTRAN V LEVEL I.

4. SUMMARY OF FEATURES

4.1 The FORTRAN V LEVEL II compiler is a two phase compiler with an optional intermediate third phase to do language dependent, machine independent optimizations.

4.2 The FORTRAN V LEVEL II compiler will provide compile-time options for conditional compilation, inclusion of source from a named file, editorial control of listings, intermix of CAL and FORTRAN, and run-time and flow tracing.

4.3 FORTRAN 5 Level II embodies several significant language extensions which include:

Address mode variables and constants
Hexadecimal constants
Mixed Mode arithmetic
ENCODE/DECODE statements
Character string variables and constants with substring and concatenation operators

4.4 The FORTRAN V Level II compiler will be a three phase compiler. Phase 1 will be the lexical syntactic analyzer producing a symbol table, a label table, an intermediate language matrix, and a DO-table stack. Phase 2 is an optional phase which will do language dependent, machine independent optimizations. Common subexpression analysis, constant computations, and locally constant expressions out of DO-loops will be handled by this phase. The third phase will be the code generation phase. It will perform machine dependent optimizations, register allocation and special machine checks for optimization purposes.

5. DETAILED DESCRIPTION

5.1.1 This functional specification establishes:

(1) The form of a program written in the FORTRAN V Level II language.

(2) The form of writing input data to be processed by such a program operating on INTERDATA computers.

(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 INTERDATA computer systems.

5.1.2 Notation Used in This Functional Specification.

In describing the form of FORTRAN statements or constructs, the following conventions and symbols are used:

(1) Special characters from the FORTRAN character set, upper case letters, and upper case words are to be written exactly as shown, except where otherwise noted.

(2) Lower case letters or words indicate general entities for which specific entities must be substituted in actual statements.

(3) Brackets, [ ] are used to indicate an optional item, or to group optional items.

(4) An ellipsis,..., indicates that the preceding optional item(s) may be repeated more than once in succession. Ellipses are also used to indicate missing items where the context is obvious, for example, digits are 0,1,2,...,9.

(5) Blanks are used to improve readability, but unless otherwise noted have no significance.

An example illustrates the above. Given a description of the form of a statement as:

```
CALL sub[(a ,a ...)]
```

then the following forms are allowed:

```
CALL sub
CALL sub (a)
CALL sub (a,a)
```

```
CALL sub (a,a,a)
etc.
```

When writing an actual statement, specific entities are substituted for sub and each a: for example, CALL ABCD(X,1.0)

5.2 FORTRAN CONCEPTS

This section introduces some basic terminology and some basic concepts. A rigorous treatment of some of these terms and concepts is given in later sections. Certain conventions concerning the meaning of grammatical forms and particular words are presented. The words underlined are defined here and used throughout this functional specification.

5.2.1 Programs

An executable program consists of one main program, any number of subprograms, and any number of other external procedures. It usually is a self-contained computing procedure.

5.2.1.1 Main Program - A main program is a set of FORTRAN statements, optional comment lines, and an END line. It must not contain any FUNCTION SUBROUTINE, or BLOCK DATA statements. It can optionally be headed by a PROGRAM statement.

5.2.1.2 Subprogram - A subprogram is a set of FORTRAN statements, optional comment lines, and an END line that is headed by a FUNCTION, SUBROUTINE, or BLOCK DATA statement.

5.2.1.3 Program Unit - A program unit is either a main program or a subprogram.

5.2.2 Statements, Comments and Lines

A program unit consists of statements, optional comment lines and an END line.

5.2.2b

A statement is written in lines, the first of which is called an initial line and succeeding lines, if any, are called continuation lines. There may be a maximum of 19 continuation lines following an initial line.

5.2.2c

There is a type of line called a comment line that is not a statement and merely provides information for documentary purposes. It starts with a C in Column 1.

5.2.2d

There is a type of line called a system option line. This line begins with a $ in column 1 followed by the system option.

5.2.2e

Conditionally compiled statements are those which begin with an X in column 1 of an initial line. Such statements are ignored by the compiler until such time as a system option card controlling conditional compilation is encountered.

5.2.2f

The END line indicates the physical end of a program unit.

5.2.2.1 Classes of Statements - The statements in FORTRAN fall into two classes, executable and nonexecutable. The executable statements specify the action of the program and the nonexecutable statements describe the characteristics and arrangement of data, editing information, statement functions, the kind of subprograms, and entry points within subprograms.

5.2.3 Names and Options

Thy syntactic parts of a statement are names, constants, operators, and special characters. Names are used to reference data and procedures. Operators, including the imperative verbs, usually specify action upon named data.

5.2.3.1 Symbolic Names - A symbolic name consists of from one to six alphanumeric characters, the first of which must be a letter.

5.2.3.2 Array Names - An array name is a symbolic name that refers to a set of data. The number of individual data elements in the set is specified in an array declarator. The array name may sometimes be used to refer to the entire set. The array name may be qualified with a subscript to refer to a particular element of the array. The indivicual elements of an array are called array elements.

5.2.3.3 Variable Names - A variable name is a symbolic name that refers to a datum which is neither an array element nor a constant.

5.2.3.4 Expressions - The simplest expression is a constant, a variable name, an array name, or a function reference. More complicated expressions containing one or more of the above entities may be formed by using operators that express the computation to be performed.

5.3 CHARACTERS, LINES, AND EXECUTION SEQUENCE

5.3.1 FORTRAN Character Set

The FORTRAN character set consists of twenty-six letters, ten digits, and fifteen special characters.

5.3.1.1 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

5.3.1.2 A digit is one of the ten characters:

0 1 2 3 4 5 6 7 8 9

A string of digits is interpreted in the decimal base number system when a number system base interpretation is appropriate. The hexadecimal base number system is written as X'h_1,...h_4' or Y'h_1,...,h_8' where the h_i are digits.

5.3.1.3 Special Characters - A special character is one of the sixteen characters.

Character	Name of Character
	Blank
=	Equals
+	Plus
-	Minus
*	Asterisk
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Decimal Point
'	Apostrophe
:	Colon
<	Less Than
>	Greater Than
;	Semicolon
@	At Sign

5.3.1.4 Blank Character - With the exception of the uses in Hollerith and character strings, a blank character within a program unit has no meaning and may be used to improve the appearance of the program, subject to the restriction on the number of consecutive continuation lines.

5.3.2 Lines

A line in a program unit is a string of 72 characters. All characters must be from the FORTRAN character set except in

comments, Hollerith strings, or character strings.

5.3.2b

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. Lines are ordered by the sequence in which they are presented to the processor. Thus a program unit consists of a totally ordered set of characters.

5.3.2.1 Comments - The letter C in column 1 of a line designates that line as a comment line. A semicolon after any FORTRAN statement designates the remainder of that line as a comment. Any character capable of being represented in the processor may appear in a comment. A comment does not affect the executable program in any way and is available as a convenience for the user.

5.3.2.1b

A comment line must be immediately followed by an initial line, another comment line, or an END line. A comment line may not be followed by a continuation line. Comment lines may precede the initial line of the first statement of any program unit.

5.3.2.2 An END line is any line that has the character blank in columns 1 through 6 and the characters E, N, and D once each, and in that order, in columns 7 through 72, preceded by interspersed with, or followed by the character blank. The END line contains no other characters in columns 1 through 72 except perhaps a semicolon followed by a comment. The END line indicates to the processor the end of the written description of a program unit. Every program unit must physically terminate with an END line.

5.3.2.3 Initial Line - An initial line is any line that is neither a comment line nor an END line and contains the digit 0 or the character blank in column 5. Column 1 may contain an X for a conditionally compiled line, otherwise columns 1 through 5 or columns 2 through 5 contain a statement label or the character blank.

5.3.2.4 Continuation Line - A continuation line is any line that is not a comment line and contains any character of the FORTRAN character set other than the digit 0 or the character blank in column 6. A continuation line may follow only an initial line or another continuation line and there must not be more than nineteen consecutive continuation lines. Columns 1 through 5 of a continuation line may contain any characters of the FORTRAN character set, except that column 1 must not contain a C or an X or a $.

5.3.2.5 System Option Lines - System option lines start with a $ in column 1 followed by the system option command.

5.3.2.5b

Mixing of Assembly Language with FORTRAN - The compiler processes statements as FORTRAN source code until a $ASSM statement is encountered. Each source statement read subsequent to $ASSM is passed directly to the intermediate text without any analysis by the compiler, other than checking for other option statements. This continues until a $FORT statement is encountered.

5.3.2.5c

Conditional Compilation - $COMP, $NCMP are used to control conditional compilation. Source statements flagged with an X in column are not compiled until a $COMP statement is encountered. Then they are compiled until either a $NCMP option a FORTRAN END statement is encountered.

5.3.2.5d

Flow Tracing and Run-time Value Tracing - Three types of run-time trace options will be available for use with the FORTRAN V level II compiler. The first is used in tracing selected variables, the second is used in tracing all variables within a specified area, and the third is an unconditional trace of all variables within a program unit. Trace options statements may not appear in BLOCKDATA subprograms.

A $TRCE statement used for item tracing, specifies a list of variable names and/or array names. The format for this type of statement is:

```
col
1     7
$TRCE X1,X2,...,Xn
```

where: X is any variable or array name. When any variables or array elements listed in a previously encountered $TRCE statement become redefined by an arithmetic statement, coding is inserted causing a line of trace information to be printed on LU6 at run time.

A $TRCE statement used for area tracing specifies a single statement number and has the format:

```
col
1     7
$TRCE n
```

where: n is any statement number not yet defined at the time the $TRCE statement is processed. This type of trace inserts

coding that causes the results of all arithmetic expressions that follow the $TRCE statement, up to the statement specified in the $TRCE statement, to be printed on LU 6. In addition to tracing all arithmetic statements within the trace range, all statement numbers defined within the range also cause coding to be generated that prints a line of trace information indciating the statement number encountered. An area $TRCE statement should not be placed within the trace range of another area $TRCE statement.

Insertion of a $TRCE statement with no argument list of identifier names or statements causes coding to be generated to trace all variables and labelled statements until an END or $NTRE statement is encountered.

Insertion of a $NTRE statement unconditionally terminates all tracing until a subsequent $TRCE statement is encountered. If no $TRCE options are specified, no trace code is generated; i.e., the $NTRE option is assumed.

5.3.2.5e

$TITL System Option - This option causes the contents of columns 7 through 72 of the $TITL statement to be used as a heading along with the page number on the compiler's list output. $TITL statements may be used anywhere within the source program. The list output device is ejected to top-of-form and a new title is printed each time a $TITL statement is encountered.

5.3.2.5f

$EJCT System Option - This option causes the compiler to eject the list output device to top-of-form, print the current title and next page number, and continue.

5.3.2.5g

$TEST - This option causes code to be generated subsequent to the computation of array indices to test for index values that are out of range. The array range check is made only for arrays which have fixed dimensions; adjustable dimension arrays in subroutines are not tested.

5.3.3 Statements

The statements of the FORTRAN language are described in later sections and are used to form program units. Each statement is written in columns 7 through 72 of an initial line and possibly as many as nineteen continuation lines. Thus a statement may not contain more than 1320 characters. Except as part of a logical IF statement no statement can begin on

a line that contains any part of the previous statement.

5.3.3b

Blank characters within a statement do not change the interpretation except when they appear within the datum strings of Hollerith constants, character constants or the Hollerith or character field descriptors in FORMAT statements. However, blank characters do count as characters in the limit of 1320 characters in any one statement. A statement of all blank characters is not permitted.

5.3.4 Statement Label

Statement labels provide a means of referring to individual statements. Any statement may be labelled, but only labeled executable statements and FORMAT statements may be referred to. A statement label consists of one to five decimal digits. 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.

5.3.4b
The form of a statement label is the same as that of an unsigned, nonzero, integer constant; however, a statement label is not an integer constant. The value of the integer represented is not significant. Blanks and leading zeros are not significant in distinguishing between statement labels. NOTE that there are 99999 unique statement labels.

5.3.5 Order of Statements and Lines

If a PROGRAM statement appears in a main program, it must be the first statement. The first statement of a subprogram must be either a FUNCTION, SUBROUTINE, or BLOCKDATA statement.

5.3.5b
Within a program unit: (1) FORMAT statements may appear anywhere and ENTRY statements may appear anywhere except within ranges of DO-loops; (2) all specification statements must precede all executable statements, DATA statements, and statement function definitions; (3) all statement function definitions must precede all executable statements and (4) DATA statements may appear anywhere after the specification statements.

5.3.5c
Within the specification statements of a program unit, IMPLICIT statements must precede all other specification statements except PARAMETER statements. A PARAMETER statement must precede all other statements containing the symbolic names

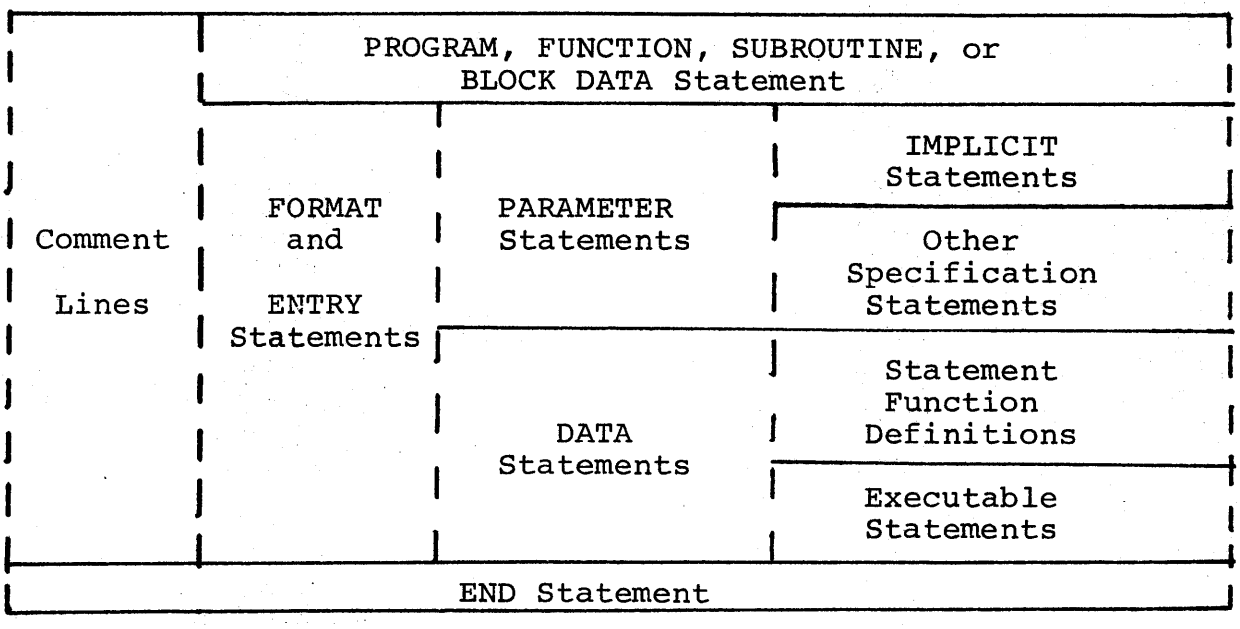
of constants that appear in that particular PARAMETER statement. If a variable appears in an adjustable dimension declarator expression or in a character length expression, the type ot that variable may not be specified in a subsequent type-statement.

5.3.5d
Every program unit must have an END line as its last line.

5.3.5e

Figure 1.

Required Order of Statements and Lines



5.3.5f
Figure 1 is a pictorial representation of the required order of statements for a program unit as described in this section. Vertical lines delineate varieties of statements which may be interspersed. For example, FORMAT statements may be interspersed with statement function definitions and executable statements. Horizontal lines delineate varieties of statements which may not be interspersed. For example, statement function definitions may not be interspersed with executable statements.

5.3.6 Normal Execution Sequence

Normal Execution Sequence is the execution of executable statements in the order in which they appear in a program unit. Execution of an executable program begins with the execution of the first executable statement of the main program. When a subprogram is referenced by the name of the subprogram, execution starts with the execution of the first executable statement of that subprogram. When a subprogram is referenced by an entry name, execution starts with the first executable statement following the corresponding ENTRY statement.

5.3.6b
Statements that may cause other than the normal execution sequence are:

(1) GO TO
(2) Arithmetic IF
(3) RETURN
(4) STOP
(5) Input/Output statements containing an error specifier or an end-of-file specifier
(6) A logical IF statement containing any of the above forms
(7) The terminal statement of a DO-loop

5.3.6c
The execution sequence is not affected by the appearance of nonexecutable statements or comment lines between executable statements.

5.3.6d
A program unit may not contain an executable statement that can never be executed.

5.3.6e
If the execution sequence attempts to proceed beyond the last executable statement of a main program, the effect is the same as the execution of a STOP statement. If the execution sequence attempts to proceed beyond the last executable statement of a procedure subprogram, the effect is the same as the execution of a RETURN statement.

5.3.6f
In the execution of an executable program, a procedure subprogram may not be referenced twice without the execution of a RETURN statement or the effect of executing a RETURN statein that procedure having intervened.

5.4 DATA TYPES, CONSTANTS, AND STORAGE

5.4.1 Data Types

The ten types of data are:

(1) integer
(2) integer *2
(3) real
(4) double precision
(5) complex
(6) logical
(7) character
(8) Hollerith
(9) bit
(10) address

Each type is different and may have a different internal representation. The type may affect the interpretation of the operations with which a datum is involved.

5.4.1.1 Data Type of a Name - The name employed to identify a datum or function also identifies its type. A symbolic name representing a variable, an array, or a function (except generic functions) must have only a single type for each program unit. Once identified with a particular type in a program unit, a specific name implies that type for any usage of that symbolic name that requires a type throughout that program unit.

5.4.1.2 Type Rules for DATA and Procedure Identifiers - A symbolic name that identifies a variable, an array, an external function (except basic external and generic function), a function entry, or a statement function may have its type specified in a type statement as integer, integer *2, real, double precision, complex, logical, character, bit or address. 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 and any other letter implies type real, unless an IMPLICIT statement is used to change the default implied type.

5.4.1.2b
The data type of an array element name is the same as its array name.

5.4.1.2c
The data type of a function determines the type of the datum it supplies to an expression in which that function is referenced.

5.4.1.2d
A symbolic name that identifies an intrinsic function or a basic external function, when it is used to identify a function in Table 3 or 4 has a type as specified in the corresponding table. An explicit type-statement is not required. The generic function names (Table 5) do not have a predetermined type; the result of a generic function reference assumes a type that depends on the type of the argument.

5.4.1.2e
In a program unit in which an external function is referenced, the type of the function is determined in the same manner for variables and arrays. The type of a function subprogram is specified either implicitly by its name, explicitly in the FUNCTION statement, or explicitly in a type-statement. For a function entry name, type is specified either implicitly by the name or explicitly in a type statement.

5.4.1.2f
If a generic function name appears in a type-statement within a program unit, that name loses its generic property within that program unit.

5.4.1.2g
A symbolic name that identifies a subroutine or subroutine entry, common block, main program, or block data subprogram has no data type.

5.4.1.2h

There exists no mechanism to identify a symbolic name with the Hollerith data type. Thus Hollerith data, other than constants, are identified under the guise of a name of any other type except character.

5.4.2 Constants

The value of a constant does not change.

5.4.2.1 Data Type of a Constant - The form of the string representing a constant defines both the value and the data type. A PARAMETER statement allows a constant to be given a symbolic name. The symbolic name of a constant assumes the type implied in the form of its corresponding constant.

5.4.2.2 Signs of Constants - An unsigned constantis a constant written without a sign. A signed constant is a constant written with a plus or minus sign. An optionally signed constant is a constant that may be either signed or unsigned. Integer, integer *2, real, and double precision constants may be optionally signed constants, except where specified otherwise. A complex constant consists of a pair of optionally signed real constants.

5.4.2.3 Blanks in Constants - Blank characters occurring in a constant, except in a character or Hollerith constant, have no effect on the value of the constant.

5.4.3 Integer or Integer *2 Type

An integer or integer *2 datum is always an exact representation of an integer value. It may assume a positive, negative, or zero value. If storage is allocated for an in-

teger *2 datum, two character storage units are assigned.

5.4.3.1 Integer or Integer *2 Constant - An integer constant or integer *2 constant is written as an optional sign followed by a nonempty string of digits. The digit string is interpreted as a decimal number.

5.4.3.2 Hexadecimal Constant - A hexadecimal constant is written as X'n_1,...n_4' or Y'n_1,...,n_8'. Where: n_i are hexadecimal digits. They may be used in place of integer constants.

5.4.4 Real Type

A real datum is a processor approximation to the value of a real number. The degree of approximation is five decimal digits. It may assume a positive, negative, or zero value. If storage is allocated for a real datum, one storage unit is allocated.

5.4.4.1 Basic Real Constant - A basic real constant is written as an optional sign, an integer part, a decimal point, and a fractional part in that order. Both the integer part and the fractional part are strings of digits; either one of these strings may be omitted but not both. A constant may be written with more digits than the compiler will use to approximate the value of that constant. A basic real constant is interpreted as a decimal number.

5.4.4.2 Real Exponent - A real exponent is written as the letter E followed by an optionally signed integer constant. A real exponent denotes a power of ten to be multiplied by the constant written immediately preceding it.

5.4.4.3 Real Constant - A real constant is written as a basic real constant, a basic real constant followed by a real, exponent, or an integer constant followec by a real exponent.

5.4.4.3b
The value of a real constant that contains a real exponent is the value of the constant which precedes E of the real exponent multiplied by the power of ten indicated by the integer written following the E in the real exponent.

5.4.5 Double Precision Type

A double precision datum is a compiler approximation to the value of a real number. This degree of approximation is to fifteen decimal digits. A double precision datum may assume a positive, negative, or zero value. If storage is allocated for a double precision datum, two storage units are allocated.

5.4.5.1 Double Precision Exponent - A double precision exponent is written and interpreted identically to a real exponent except that the letter D is written instead of the letter E.

5.4.5.2 Double Precision Constant - A double precision constant followed by a double precision exponent, or as an integer constant followed by a double precision exponent.

The value of a double precision constant that contains a double precision exponent is the value of the constant which precedes D of the double precision exponent multiplied by the power of ten indicated by the integer written following the D in the double precision exponent.

5.4.5 A complex datum is a processor approximation to the value of a complex number. The representation of the approximation is in the form of an ordered pair of read data. The first of the pair represents the real part and the second represents the imaginary part of the complex datum. Each part has the same degree of approximation as for a real datum. If storage is allocated for a complex datum, two storage units are allocated.

5.4.6.1 Complex Constant - A complex constant is written as an ordered pair of optionally signed real constants, separated by a comma, and enclosed within parentheses. The first real constant of the pair is the real part of the complex constant and the second real constant of the pair is the imaginary part.

5.4.7 Logical Type

A logical datum may assume only the values true or false. If storage is allocated for logical datum, one storage unit is allocated.

5.4.7.1 Logical Constant - A logical constant is written as .TRUE. or .FALSE.; the value of .TRUE. is true and the value of .FALSE. is false.

5.4.8 A character datum is a string of characters. This string may consist of any characters capable of representation in the processor. The blank character is valid and significant in a character datum. The length of a character datum is the number of characters in the string. If storage is allocated for character datum, one character storage unit is allocated for each character in the string.

5.4.8b
Each character in the string has a character position that is consecutively numbered 1, 2, 3, etc. The number indicates the sequential position of a character in the string, starting at the left and proceeding to the right.

5.4.8.1 Character Constant - A character constant is written as a contiguous string of characters; the first and last characters must be apostrophes that delimit the constant.

The delimiting apostrophes are not part of the datum represented by the constant. An apostrophe within the datum string is written as two consecutive apostrophes with no intervening blanks.

5.4.8c
The length of a character constant is the number of characters written between the delimiting apostrophes, except that each pair of consecutive apostrophes counts as a single character. The delimiting apostrophes are not counted. The length of a character constant must be greater than zero.

5.4.9 Hollerith Type

A Hollerith datum is a string of characters. This string may consist of any characters capable of representation in the processor. The blank character is significant in a Hollerith datum.

5.4.9.1 Hollerith Constant - A Hollerith constant is written as a nonzero, unsigned integer constant n followed by the letter H, followed by exactly n characters which comprise the Hollerith datum. Any n characters capable of representation in the processor may follow the H.

5.4.9.1b
In a Hollerith constant, blanks are significant only in the n characters following the letter H.

5.4.9.2 Restrictions on the Use of Hollerith Constants - A Hollerith constant may be written only in the argument list of a CALL statement and in the DATA statement.

5.4.10 Bit Type

A bit datum may assume only the values of one or zero. If storage is allocated for a bit datum, one bit is allocated and then storage is rounded up to the nearest character storage unit.

5.4.10.1 Bit Constants - A bit constant may be represented by a hexadecimal constant.

5.4.11 Address Constants

Address constants are written in the form A'nam' where nam is an identifier associated with a variable, array, array element, or statement label. The effect of an address constant in an expression is to use the address of the named identifier or label as opposed to its value. Address constants may only appear in expressions involving other address constants, address mode variables integer constants or integer variables.

5.4.12 Storage

Storage may be allocated for a datum. The amount of storage allocated depends on the type of the datum.

5.4.12.1 Storage Unit - If storage is allocated to a datum, one storage unit is allocated if it is integer, real, logical, or address, and two logically consecutive storage units are allocated if the datum is double precision or complex.

5.4.12.2 Character Storage Unit - If storage is allocated to a character datum of length n logically consecutive character storage units are allocated. There are four character storage units for each storage unit. A n INTEGER*2 datum requires two character storage units. A bit datum requires one character storage unit.

5.5 ARRAYS

An array is a named and ordered set of data. An array element is one member of the set of data. An array name is the symbolic name of the array. An array element name is an array name qualified with a subscript.

5.5b
Where it is permitted, the entire ordered set of data is identifiable simply by use of the array name. A single member of the set is identifiable by use of an array element name.

5.5c
By treating a set of data as an array and using an array element name, it is possible to identify and define or reference a particular set of data based on the subscript value. In different executions of a statement containing an array element name, the array element name identify different array elements depending on the value of the subscript during each of the executions.

5.5.1 Array Declarator

An array declarator specifies a symbolic name that identifies an array within a program unit, and specific certain properties of the array.

5.5.1 Form of an Array Declarator - An array declarator is of the form:

a (d [,d] ...)

where: a ,called the declarator name, is the symbolic name of the array.

d is a dimension declarator

The number of dimensions of the array is the number of dimension declarators in the array declarator. The minimum number of dimensions is one and the maximum is seven.

5.5.1.1.1 Form of a Dimension Declarator - A dimension declarator is of the form:

$d_1 : d_2$

where: d_1 is the lower dimension bound

d_2 is the upper dimension bound

The upper and lower bounds are arithmetic expressions, in which all constants and variables are integer entities. Excluded from dimension bound expressions are function and array element references. Integer variables may appear in dimension bound expressions only in adjustable array declarators.

5.5.1.1.2 Value of Dimension Bounds - If only the upper bound is specified, the value of the lower bound is one.

5.5.1.1.2b
If the lower dimension bound is specified, its value may be negative, zero, or positive; its value may be one. The value of the upper dimension bound must be greater than or equal to the value of the lower dimension bound.

5.5.1.2 Kinds and Occurrences of Array Declarators - Each array declarator is either a constant array declarator or an adjustable array declarator. A constant array declarator is an array declarator in which each of the dimension bound expressions is a constant expression. An adjustable array declarator is an array declarator in which one or more of the dimension bound expressions contains a variable.

5.5.1.2b
Any variable that appears in an adjustable dimension declarator expression must appear as a dummy argument or as an entity in a common block in the program unit containing

5.5.1.2c
Each array declarator is either an actual array declarator or a dummy array declarator. A dummy array declarator is an array declarator in which the declarator name is a dummy argument or in common. An actual array declarator is an array declarator which is not a dummy array declarator.

5.5.1.2d
Each actual array declarator must be a constant array declarator. An actual array declarator is permitted to appear in a COMMON statement, a DIMENSION statement, or a type-statement.

5.5.1.2e
A dummy array declarator is permitted to be either a constant array declarator or an adjustable array declarator. A dummy array declarator is permitted to appear in a DIMENSION statement or a type-statement but not in a COMMON statement.

5.5.1.3 Effect of an Array Declarator - The presence of an array declarator in a program unit specifies that the declarator name is an array name within that program unit. Note that this enables the compiler to distinguish between constructs which are identical in form, but which identify different entities (for example, an array element name and a function reference).

5.5.2 Properties of an Array

The array declarator specifies the following properties of the array: the number of dimensions of the array, the size and bounds of each dimension, the number of array elements, and the array element ordering.

5.5.2b
The properties of an array in a program unit are determined by the array declarator for that array name in that program unit. The same array name may have different properties in different program units.

5.5.2.1 Data Type of an Array and an Array Element - An array name has a data type. An array element name has the same data type as the array name.

5.5.2.2 Dimensions of an Array - The number of dimensions of an array is equal to the number of dimension declarators in the array declarator. The size of a dimension is the value:

$$d_2 - d_1 + 1$$

where: d_1 is the value of the lower dimension bound

d_2 is the value of the upper dimension bound

Note that if the value of the lower dimension bound is one, the size of the dimension is d_2.

5.5.2.3 Number of Array Elements - The number of elements in an array is equal to the product of the sizes of the dimensions of the array indicated by the array declarator for that array name. The size of an array is equal to the number of elements in the array.

5.5.2.4 The elements of an array are ordered sequentially according to the array element subscript value. The first element of the array has a subscript value of one; the second element has a subscript value of two; the last element has a subscript value equal to the size of the array. Whenever the use of an entire array is indicated by an array name unqualified by a subscript the elements of the array are taken according to the array element ordering.

5.5.3 Array Element Names

An array element name is of the form:

a (s [,s] ...)

where a is the array name

(s [,s] ...) is a subscript

s is a subscript expression

The number of subscript expressions must be equal to the number of dimensions in the array declarator for the array name which the subscript qualifies, except in an EQUIVALENCE statement.

5.5.4 Subscripts

Throughout this functional specification, the term subscript includes the parenthesis that delimit the list of subscript expressions. A subscript has a subscript value which determines which element of the array is identified by the array element name.

5.5.4.1 Form of a Subscript - A subscript is of the form:

(s [,s] ...)

where s is a subscript expression

The subscript value is specified in Table 1.

5.5.4.2 Subscript Expression - A subscript expression is any integer, real, or double precision expression. A subscript expression may contain array element references and function references. Note that a restriction in the evaluation of expressions prohibits certain side effects. In particular, evaluation of a function may not alter the value of any other subscript expressions within the same subscript.

5.5.4.2b
If the subscript value expression is of type other than integer, the value is converted to type integer according to the rules for arithmetic assignment statements before computing the value of the subscript.

5.5.4.2c
The value of each subscript expression must be greater than or equal to each corresponding lower dimension bound for the program unit containing the subscript. The value of each subscript may exceed the corresponding upper dimension bound. However, the subscript expression values are constrained such that the subscript value must be greater than or equal to one

and must be less than or equal to the number of subscripts in the array.

5.5.4.3 Subscript Value - The subscript value is specified in Table 1. The value of the subscript determines which array element is identifed by the array element name. The subscript value depends on the values of the subscript expressions in the subscript and on the dimensions of the array specified in the array declarator for that array in that program unit. If the subscript value is K, then the Kth element of the array is identified.

Table 1

Subscript Value

Dimensions	Dimension Declarator	Subscript	Subscript Value
1	$(d_{11}:d_{12})$	(S_1)	$1 + (S_1-d_{11})$
2	$(d_{11}:d_{12},d_{21}:d_{22})$	(S_1,S_2)	$1 + (S_1-d_{11}) + (S_2-d_{21})*d_1$
3	$(d_{11}:d_{12},d_{21}:d_{22},d_{31}:d_{32})$	(S_1,S_2,S_3)	$1 + (S_1-d_{11}) + (S_2-d_{21})*d_1 + (S_3-d_{31})*d_2*d_1$
. . .			
n	$(d_{11}:d_{12},\ldots,d_{n1}:d_{n2})$	$(S_1,\ldots,S_n)$	$1 + (S_1-d_{11}) + (S_2-d_{21})*d_1 + (S_3-d_{31})*d_2*d_1 + \ldots + (S_n-d_{n1})*d_{n-1}*d_{n-2}*\ldots*d_1$

Notes for Table 1:

(1) $1 \geq n \geq 7$
(2) di_1 is the value of the lower bound of the ith dimension
(3) di_2 is the value of the upper bound of the ith dimension
(4) If only the upper bound is specified, then $di_1 = 1$
(5) si is integer value of ith subscript expression
(6) di is size of ith dimension
$di = di_2 - di_1 + 1$
If the value of the lower bound is 1, then $di = di_2$

5.5.4.3c

A consequence of Table 1 is that a subscript of the form $(d_{11},\ldots,d_{n1})$ has a subscript value of one and identifies the first element of the array. A subscript of the form $(d_{12},\ldots,d_{n2})$ identifies the last element of the array; its subscript value is equal to the number of elements in the array.

5.5.4.3d
Note that the subscript value and the subscript expression value may not be the same, even for a one dimensional array. For example, given:

```
DIMENSION A(-1:10),B(10,10)
A(2) = B(1,2)
```

then A(2) identifies the fourth element of A, the subscript is (2) with a value of four, the subscript expression is 2 with a value of two, B(1,2) identifies the eleventh element of B, the subscript is (1,2) with a value of eleven, and the subscript expressions are 1 and 2 with values of one and two.

5.5.5 Dummy Arrays

A dummy array is an array for which the array declarator is a dummy array declarator. A dummy array is permitted only in a function or subroutine subprogram.

5.5.5b
At the time of execution of a reference to a subprogram that contains a dummy array, the actual argument corresponding to the dummy array name must be either an array name or an array element name. If the actual argument is an array name, then the size of the dummy array must not be greater than the size of the actual argument array. If the actual argument is an array element name with a subscript value of p in an array size n, then the size of the dummy array must not exceed n+1-p. Each dummy array name must be associated through one or more levels of external procedure references with either an actual array name or an actual array element name.

5.5.5.1 Adjustable Arrays and Adjustable Dimensions - An adjustable array is an array for which the array declarator is an adjustable array declarator. In an adjustable array declarator, those dimension declarators that include a variable name are called adjustable dimensions.

5.5.5.1b
An adjustable array declarator must be a dummy array declarator. At least one dummy argument list of the subprogram must contain the name of the adjustable array. A variable name that appears in a dimension declarator expression must also appear as a name either in every dummy argument list or in a common block in that subprogram.

5.5.5.1c
At the time of execution of a reference to a subprogram that contains an adjustable array, each actual argument that corresponds to a dummy argument that appears in an adjustable dimension expression and each variable in common that appears in an adjustable dimension expression must have a defined integer value. The values of those dummy arguments or variables in common, together with any constants that appear in that adjustable dimension expression, determine the size of the corresponding adjustable dimension for that execution of that subprogram. The sizes of the adjustable dimensions and of any constant dimensions that appear in an adjustable array declarator determine the number of elements in that array and the array element ordering. The execution of different references to a subprogram or different executions of the same reference determine possibly different properties (size of dimensions, dimension bounds, number of elements, and array element ordering) for each adjustable array in that subprogram. These properties depend on the values of any actual arguments and variables in common that are referenced in the adjustable dimension expressions in that subprogram.

5.5.5.1d
During execution of a subprogram containing an adjustable array, the array properties of dimensionsize, upper and lower dimension bounds, and array size (number of elements in the array) do not change. However, the variables involved in an adjustable dimension may change or become undefined during execution of the subprogram but such a change or undefinition does not affect the above mentioned properties.

5.5.6 Use of Array Names and Array Element Names

For a symbolic name to be an array name in a program unit, it must appear as a declarator name in that program unit. Only one array declarator for an array name is permitted in a program unit.

5.5.6b
A symbolic name that is used to identify an array or an array element may be used also in the same program unit to identify a common block but must not be used in the same program unit to identify any other entity.

5.5.6c
In a program unit, each appearance of an array name must be in an array element name except in 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

(6) as the format identifier in an input/output statement

(7) in an EQUIVALENCE statement

(8) in a DATA statement

(9) in an array declarator. Note that although the form of an array declarator is identical to one of the forms of an array element name, an array declarator is not an array element name.

(10) in a SAVE statement

(11) As the internal unit identifier of a storage file in an input/output statement

5.5.6d
Whenever an array name unqualified by a subscript is used to designate the whole array, the appearance of the array name implies that the number of values to be processed is equal to the number of elements in the array and that the elements of the array are taken in consecutive sequential order.

5.6 EXPRESSIONS

This section gives the formation and evaluation rules for arithmetic, relational, character, and logical expressions. An expression is formed from operands and operators.

5.6.1 Arithmetic Expressions

An arithmetic expression is used to express a numeric computation. Evaluation of an arithmetic expression produces an arithmetic value.

5.6.1b
The simplest form of an arithmetic expression is an unsigned constant, a variable reference, an array element reference, or a function reference. More complicated arithmetic expressions may be formed by using one or more arithmetic operands together with arithmetic operators and parentheses. The order in which the indicated operations are performed in the evaluation of an arithmetic expression depends on the appearance of parentheses, the priority of the operators, and the data type of the operands.

5.6.1.1 Arithmetic Operators - The five arithmetic operators are:

Operator	Representing
**	exponentiation
/	division
*	multiplication
-	subtraction
+	addition

Each of the arithmetic operators is used to express a numeric computation to be performed on arithmetic operands. Each of the operators **, /, and * operates on a pair of operands and is written between the two operands. Each of the operators + and - either:

(1) operates on a pair of operands and is written between the two operands, or

(2) operates on a single operand and is written preceding that operand.

5.6.1b
The operation identificed by each of the arithmetic operators in each form of use is as follows:

Use of Operator	Operation
X_1 **X_2	Exponentiate X_1 to the power X_2
X_1 /X_2	Divide X_1 by X_2
X_1 *X_2	Multiply X_1 and X_2
X_1 -X_2	Subtract X_2 from X_1
-X_2	Subtract X_2 from zero
X_1 +X_2	Add X_2 and X_1
+X_2	Add zero and X_2

where: X_1 denotes the operand to the left of the operator

X_2 denotes the operand to the right of the operator.

When an integer operand is divided by an integer operand, any fractional part in the quotient is discarded. The result is the nearest integer whose magnitude does not exceed the magnitude of the mathematical value of the quotient. For example, the value of the expression (-8)/3 is -2.

5.6.1.1c
The priority of the operators is implied in the rules for the formation and evaluation of arithmetic expressions. In general, the order in which the indicated operations are performed de-

pends on the priority of the operators appearing in an expression, unless the order is changed by the use of parentheses (6.1.3). The priority of each arithmetic operator is:

Operator	Priority
**	highest
* and /	intermediate
+ and -	lowest

For example, in the expression

```
-A ** 2
```

the exponentiation operator (**) has a higher priority than the subtraction operator (-); the value of the above expression is the same as the value of the expression

```
-(A**2)
```

5.6.1.2 Arithmetic Operands - An arithmetic expression and the operands in an arithmetic expression must identify values of one of the arithmetic data types: integer, real, double precision, or complex. The operands may all be of the same type or may be of different types. A mixed-type arithmetic expression is an arithmetic expression that contains operands of different data types. Except for a value raised to an integer power, the evaluation of a mixed-type arithmetic expression includes an implied conversion of data type of some of the operands as that evaluation proceeds. The order of evaluation of integer expressions and of mixed-type expressions is more restricted than the order of evaluation of real, double precision, or complex expressions.

5.6.1.2b
The arithmetic operands are:

(1) primary
(2) factor
(3) product
(4) term
(5) arithmetic expression

5.6.1.2.1 Primaries - A primary is one of the forms:

(1) unsigned constant
(2) variable reference
(3) array element reference
(4) function reference
(5) arithmetic expression enclosed in parentheses

5.6.1.2.1b
The data type of each of the first four kinds of primaries is identified by the form of the constant or the name of the datum or function (Section 4). The data type of a primary of the form (5) is the data type of the value of the expression as defined in Tables 6.1 and 6.2, and is independent of the context in which that primary appears. For example, in the expression

```
(5/2) * A ** (5/2)
```

the primary (5/2) identifies the integer value 2 in each occurrence.

5.6.1.2.2 Factors - a factor is of one of the forms:

(1) primary
(2) primary ** factor

Form (2) requires that exponentiation operations are performed from right to left within a factor that contains two or more exponentiation operators. For example, the factor

```
2 ** 3 ** 2
```

is equivalent in value to the factor

```
2 ** (3**2)
```

5.6.1.2.2b
The data type and value of a factor that contains exponentiation is shown in Table 6.2.

5.6.1.2.3 Products - A product is of one of the forms:

(1) factor
(2) product * product

Form (2) permits the associative law of multiplication to be used by a processor to group the factors in the evaluation of any product that contains two or more multiplication operators. For example, the value of the proucts

```
A * B * C * D
```

is the same as the value of any of the following products:

```
((A*B) * C) *D
A * (B * (C*D))
(A*B) *(C*D)
(A * (B*C)) *D
A * ((B*C) *D)
```

The commutative law of multiplication may be used by a processor to order the factors within a product. For example, the value of the product

```
A * B * C
```

is the same as the value of any of the following products:

```
A * C * B
B * A * C
B * C * A
C * A * B
C * B * A
```

5.6.1.2.3b
The data type and value of a product that contains two or more factors is shown in Table 6.1.

5.6.1.2.4 Terms - a term is of one of the forms:

(1) product
(2) term/factor
(3) term * product
(4) term * term

The commutative and associative laws of multiplication may be used by a processor to order the factors in a term.

5.6.1.2.4b
Forms (1), (2), and (3) may be used by a processor in the evaluation of any term. Form (4) of a term may not contain integer division in the right operand. For example, in the expression

```
B/A * I/J
```

is written, it may not be interpreted by the processor to mean

```
(B/A) * (I/J)
```

The only valid interpretation is by use of form (2) and form (3) and is

```
((B/A) * I)/J
```

Note that the restriction placed on the use of form (4) means that integer terms and some mixed-type terms must be evaluated in effect from left to right.

5.6.1.2.4c
In a term that contains two or more successive division operators, the division operations must be performed in left to right order. For example, the value of the term

8. / 4. / 2.

is one.

5.6.1.2.4d
Form (3) may be used to evaluate the term

A / B * C * D

The value of the above term is the same as the value of the term

(A/B) * (C*D)

5.6.1.2.4e
Form (4) may be used to evaluate the term

A * B/C

in the following order:

A * (B/C)

However, if the factors are of integer type, only form (2) may be used. The term must be evaluated in the following order:

(A*B) / C

Note that

the value of (8*7) / 4 is 14 but
the value of 8 * (7/4) is 8

5.6.1.2.4f
The data type and value of a term is shown in Table 6.1.

5.6.1.2.5 Arithmetic Expressions - An arithmetic expression is of one of the forms:

(1) term
(2) + term
(3) - term
(4) arithmetic expression + term
(5) arithmetic expression - term

Each arithmetic expression is mathematically equivalent to an arithmetic expression that is a sum of terms. For example, the value of the expression

A - B + C

is the same as the value of the expression

A + (-B) + C

The associative law of addition may be used by a processor to associate the terms to be summed. For example, the value of the expression

```
A + B - C
```

is the same as the value of either of the following expressions:

```
(A+B) - C
A + (B-C)
```

The commutative law of addition may be used by a processor to order the operands. For example, the value of the expression

```
A + B - C
```

is the same as the value of any of the following expressions:

```
A - C + B
B + A - C
B - C + A
-C + A + B
-C + B + A
```

5.6.1.2.5b
Forms (4) and (5) of an arithmetic expression specify that it is formed in left to right order. However, a processor may evaluate an expression in any mathematically equivalent order that results from the use of the associative and commutative laws of addition.

5.6.1.2.5c
The data type and value of an expression is shown in Table 5.6.1.

5.6.1.3 Use of Parentheses in Arithmetic Expressions - An expression enclosed in parentheses is a primary.

5.6.1.3b
The type and value of an expression enclosed in parentheses are determined solely by applying the rules of Section 6.1 to the constituents of that expression and are independent of the contest in which the expression occurs. For example, given either of the expressions

```
R * (I/J)
(I/J) * R
```

the type and value of I/J is determined independently of the type and value of R.

5.6.1.3c
In the evaluation of an expression, a processor must not violate the integrity of an expression enclosed in parentheses. For example, a processor must not apply any of the distributive

laws of arithmetic. Thus, the expression

```
A * (B-C)
```

must not be evaluated as if it were

```
A * B - A * C
```

5.6.1.3d
Sometimes the use of parentheses is required in order to express a computation in a single statement. For example:

```
A / (B-C)
```

Note that the formation rules do not permit two consecutive operators such as A**-B or A+-B. These incorrect forms may be written using parentheses as A**(-B) and A+(-B) respectively.

5.6.1.3e
When the order of evaluation is not specified completely by the use of parentheses, a processor may revise the order of evaluation by using the commutative and associative laws as specified above. Alternative orders of evaluation are equivalent in that they all denote the same mathematical value. However, the processor approximation to the mathematical value computed by one valid order of evaluation may be different from the processor approximation computed by another valid order of evaluation. When an expression is written, additional parentheses may be included to restrict the orders of evaluation available to a processor.

Parentheses are useful for controlling the magnitude and the accuracy of intermediate values developed during the evaluation of an expression. For example,

```
A + (B-C)
```

causes B-C to be evaluated before the addition of A. Note that the inclusion of parentheses may cause an expression to denote a different value. For example, the two expressions

```
A * I / J
A * (I/J)
```

may denote different values if I and J identify factors of integer data type.

5.6.1.4 Type and Value of Arithmetic Expressions - The data type and value of the result of the arithmetic operators operating on two operands is shown in Tables 6.1 and 6.2.

5.6.1.4b
The data type of the result of the operators + or - operating on a single operand is the data type of that operand. The

value of the result of + or - operating on a single operand X_2 is equivalent to + or - operating on two operands X_1 and X_2 where the value of X_1 is zero and the type of X_1 is the type of X_2.

5.6.1.4c

Table 6.1

Type and Value of Result for +

X_1 \ X_2	I_2	R_2
I_1	$I = I_1 + I_2$	$R = FLOAT(I_1) + R_2$
R_1	$R = R_1 + FLOAT(I_2)$	$R = R_1 + R_2$
D_1	$D = D_1 + DFLOAT(I_2)$	$D = D_1 + DBLE(R_2)$
C_1	$C=C_1+CMPLX(FLOAT(I_2),0.)$	$C = C_1 + CMPLX(R_2,0.)$

X_1 \ X_2	D_2	C_2
I_1	$D = DFLOAT(I_1) + D_2$	$C=CMPLX(FLOAT(I_1),0.)+C_2$
R_1	$D = DBLE(R_1) + D_2$	$C = CMPLX(R_1,0.) + C_2$
D_1	$D = D_1 + D_2$	Prohibited
C_1	Prohibited	$C = C_1 + C_2$

Table 6.2

Type and Value of Result for **

X_1 \ X_2	I_2	R_2	D_2	C_2
I_1	$I=I_1**I_2$	$R=FLOAT(I_1)**R_2$	$D=DFLOAT(I_1)**D_2$	P
R_1	$R=R_1**I_2$	$R=R_1**R_2$	$D=DBLE(R_1)**D_2$	P
D_1	$D=D_1**I_2$	$D=D_1**DBLE(R_2)$	$D=D_1**D_2$	P
C_1	$C=C_1**I_2$	Prohibited	Prohibited	P

5.6.1.4d
Notes for Tables 6.1 and 6.2:

(1) the form of the table entry is

$$t = X_1 + X_2$$

where t is the type of the result and the expression to the right of the equals indicates the value of the result. The + operator in Table 6.1 may be replaced by -, *, or /.

(2) FLOAT, DFLOAT, DBLE, and CMPLX are the type conversion functions described in Table 5.

(3) X_1 denotes the left operand
X_2 denotes the right operand
I denotes integer data type
R denotes real data type
D denotes double precision data type
C denotes complex data type
P denotes a prohibited combination

5.6.1.4e
The order of evaluation used by a processor in a mixed-type expression may affect the data type of intermediate values developed during the evaluation of the expression. For example, in the evaluation of the expression

D + R + I

where D, R, and I identify terms of double precision, real, and integer data type respectively, the data type of the operand that is added to I may be either double precision or real depending on which pair of operands (D and R, R and I, or D and I) are added first.

5.6.1.4f
Eacept for a value raised to an integer power, Tables 6.1 and 6.2 specify that if two operands are of different data type, then the operand that differs in type from the result of the operation is performed on operands of the same type. When a value is raised to an integer power, neither operand is converted.

5.6.1.5 Restrictions on Evaluation of Arithmetic Expressions - The following are prohibited in the execution of an executable program:

(1) any arithmetic operation whose result is not mathematically defined.

(2) division by zero

(3) raising a zero-valued primary to a zero-valued or

negative-valued power

(4) raising a negative-valued primary to a real or double precision power.

5.6.1.6 Constant Expression - A constant expression is an arithmetic expression whose primaries are either an unsigned arithmetic constant or a constant expression enclosed in parentheses. Note that symbolic names of constants are allowed but variable, array element, and function references are not allowed.

5.6.1.6b
An integer constant expression is a constant expression in which all constants are integer constants. The following are examples of constant expressions:

```
3
-3
-4+4
4_(1.6**2)
```

The first three expressions are also integer constant expressions.

5.6.2 Character Expressions

Evaluation of a character expression produces a result of a type character.

6.2b
The simplest form of a character expression is a character constant, a character variable reference, a character array element reference, a character substring reference, or a character function reference. More complicated character expressions may be formed using one or more character operands together with character operators and parentheses.

5.6.2.1 Character Substrings - A character substring is used to identify, define, or reference a contiguous portion of a character variable or character array element.

5.6.2.1.1 Substring Names - A substring name is of one of the forms:

v ([e_1] : [e_2])

a ([e_1] : [e_2] , S [,S]...)

where: v is a character variable name

a (S,S...) is a character array element name

e_1 and e_2, called substring expressions, are integer,

real, or double precision expressions.

5.6.2.1.1b
e_1 specifies the leftmost character position and e_2 specifies the rightmost character position of the substring. For example, A(2:4) specifies characters in positions two through four of character variable A, and B(1:6,4,3) specifies characters in positions one through six of character array element B(4,3).

5.6.2.1.1c
The values of e_1 and e_2 must be such that:

$$1 \leq e_1 \leq e_2 \leq \underline{len}$$

where len is the length of the character variable or array element. If e_1 is omitted, a value of one is implied for e_1. If e_2 is omitted, a value of len is implied for e_2. Both e_1 and e_2 may be omitted; for example, the form v(:) is equivalent to v, and the form a(:s [,s] ...) is equivalent to a(s [,s] ...).

5.6.2.1.2 Substring Expressions - A substring expression is any integer, real, or double precision expression. A substring expression may contain array element references and function references, but a function reference may not alter the value of any entities that appear in the same substring reference.

If the value of a substring expression is not of type integer, the value is converted to type integer according to the rules for arithmetic assignment statements (Table 2).

5.6.6.2 Character Operator - The character operator is:

Operator	Representation
//	Concatenation

5.6.2.2b
The operation identified by the operator is:

Use of Operator	Operation
X_1 //X_2	Concatenate X_1 with X_2

where: X_1 denotes the operand to the left of the operator

X_2 denotes the operand to the right of the operator

5.6.2.2d
The result of a concatenation operation is a character string whose value is the value of X_1 concatenated on the right with the value of X_2 and whose length is the sum of the lengths of X_1 and X_2.

5.6.2.2c
Where there are sequential concatenation operators, concatenation proceeds from left to right. For example, the value of 'AB' // 'CD' // 'EF' is the string 'ABCDEF'.

5.6.2.3 Character Operands - A character expression and the operands in a character expression must identify values of type character.

5.6.2.3.1 Character Primaries - A character primary is of one of the forms:

(1) character constant
(2) character variable reference
(3) character array element reference
(4) character substring reference
(5) character function reference
(6) character expression enclosed in parentheses.

5.6.2.3.2 Character Expressions - A character expression is of one of the forms:

(1) character primary
(2) character primary // character primary

5.6.3 Relational Expressions

A relational expression is used to compare two arithmetic expressions or two character expressions. A relational expression may not be used to compare an arithmetic expression with a character expression.

5.6.3b
Relational expressions may appear only within logical expressions. Evaluation of a relational expression produces a result of type logical with a value of true or false.

5.6.3.1 Relational Operators - 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

5.6.3.2 Arithmetic Relational Expressions - A relational expression involving arithmetic expressions is of the form:

e_1 relop e_2

where: e_1 and e_2 are both arithmetic expressions of type integer, real, or double precision

relop is a relational operator

5.6.3.2b
A relational expression is evaluated by first evaluating each of the arithmetic operands. The numeric values of the arithmetic operands are compared as specified by the relational operator. The resulting value is true or false.

5.6.3.2c
If the two arithmetic expressions are of different types, the value of the relational expression

$$e_1 \text{ relop } e_2$$

is the value of the expression

$$((e_1) - (e_2)) \text{ relop } 0$$

where 0 (zero) is of the same type as the expression $((e_1) - (e_2))$, and relop is the same relational operator in both expressions.

5.6.3.3 Character Relational Expressions - A relational expression involving character expressions is of the form:

$$e_1 \text{ relop } e_2$$

where: e_1 and e_2 are both character expressions

relop is a relational operator

5.6.3.3b
A relational expression is evaluated by first evaluating each of the character operands. The character values of the operands are then compared as specified by the relational operator, with a resulting value of true or false. If the operands are of unequal length, the shorter operand is considered as if it were extended on the right with blanks to the length of the longer operand.

5.6.3.3c
e_1 is considered to be less than e_2 if the value of e_1 precedes the value of e_2 in the collating sequence; e_1 is greater than e_2 if e_1 follows e_2 in the collating sequence. Note that the collating sequence depends upon the processor.

5.6.4 Logical Expressions

A logical expression is formed with logical operators and logi-

cal operands. Evaluation of a logical expression produces a result of type logical with a value of true or false.

5.6.4.1 Logical Operators - The logical operators are:

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

5.6.4.2 Logical Operands - The logical operands are:

(1) logical primary
(2) logical factor
(3) logical term
(4) logical expression

5.6.4.2.1 Logical Primary - A logical primary is of one of the forms:

(1) logical constant
(2) logical variable reference
(3) logical array element reference
(4) logical function reference
(5) relational expression
(6) logical expression enclosed in parentheses

5.6.4.2.2 Logical Factor - A logical factor is of one of the forms:

(1) logical primary
(2) .NOT. logical primary

5.6.4.2.3 Logical Term - A logical term is of one of the forms:

(1) logical factor
(2) logical term .AND. logical term

5.6.4.2.4 Logical Expression - A logical expression is of one of the forms:

(1) logical term
(2) logical expression .OR. logical expression

5.6.4.3 Value of Logical Factors, Terms, and Expressions - The value of a factor involving .NOT. is shown below:

Value of X_2	Value of .NOT.X_2
true	false
false	true

The value of a logical term involving .AND. is shown below:

Value of X_1	Value of X_2	Value of X_1 .AND. X_2
true	true	true
true	false	false
false	true	false
false	false	false

The value of a logical expression involving .OR. is shown below:

Value of X_1	Value of X_2	Value of X_1 .OR. X_2
true	true	true
true	false	true
false	true	true
false	false	false

5.6.5 Evaluation of Expressions

When two operands are combined by an operator, the left or the

right operand may be evaluated first.

5.6.5b
A part of an expression need be evaluated only if such action is necessary to establish the value of the expression.

5.6.5c
If a statement contains a part of an expression that need not be evaluated, and if this part 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 function reference. For example, if the function F defines its argument X, then the statement

```
IF(.TRUE. .OR. F(X)  ) GO TO 10
```

causes X to become undefined.

5.6.5d
Any use of an array element name requires the evaluation of its subscript. The evaluation of functions appearing in a statement may not alter the value of any other entity within the statement in which the function reference appears. The evaluation of a function appearing in a statement may not alter the value of any entity in common that affects the value of any other function referenced in that statement. However, evaluation of a function in the expression e of a logical IF statement (11.5) may affect entities in the statement st that is executed when the value of the expression e is true.

5.6.5e
The data type of the expression in which a function reference appears does not affect, nor is it affected by the evaluation of the actual arguments, except when the data type of a generic function is determined by the data type of its arguments.

The data type of the expression in which a subscript appears does not affect, nor is it affected by, the evaluation of the subscript.

5.6.5f
Any variable, array element, function, or character substring referenced as a primary in an expression must be defined at the time of its use. Note that if a character string or substring is referenced, all of the characters of that string or substring must be defined at the time of reference.

5.6.5g
If a function subprogram may produce different results when referenced with the same argument list, that function must not be referenced more than once in one statement.

5.6.5h
If a function reference in a statement causes definition of an actual argument of the function, that argument or associated

entities must not appear elsewhere in the statement. For example, the statement:

```
Y = F(X) + X
```

is prohibited if the reference to F defines X.

.7. EXECUTABLE OR NONEXECUTABLE STATEMENT CLASSIFICATION

Each statement is classified as executable or nonexecutable. Executable statements specify actions and form an execution sequence in an executable program. Nonexecutable statements describe characteristics, arrangement, and initial values of data, contain editing information, define statement functions, specify the classification of program units, and specify entry points within subprograms. Nonexecutable statements are not part of the execution sequence. Nonexecutable statements may be labeled, but such statement labels may not be used to control the execution sequence.

5.7.1 Executable Statements

The following statements are classified as executable:

(1) Arithmetic, logical, ASSIGN, character, and address assignment statements.

(2) Unconditional, assigned, and computed GO TO statements.

(3) Arithmetic IF and logical IF statements.

(4) CONTINUE statement

(5) STOP and PAUSE statements

(6) DO statement

(7) READ, WRITE, and PRINT statements

(8) REWIND, BACKSPACE, ENDFILE, BACKFILE, SKIPFILE, OPEN, CLOSE, and INQUIRE statements

(9) CALL and RETURN statements

(10) END statement

(11) PUSH and PULL statements

5.7.2 Nonexecutable Statements

The following statements are classified as nonexecutable:

(1) PROGRAM, FUNCTION, SUBROUTINE, ENTRY, and BLOCK DATA statements.

(2) DIMENSION, COMMON, EQUIVALENCE, type-statements, IMPLICIT, PARAMETER, EXTERNAL, INTRINSIC, SAVE, and STACK statements.

(3) DATA statement

(4) FORMAT statement

(5) Statement function definition statement.

5.8 SPECIFICATION STATEMENTS

There are ten kinds of specification statements:

(1) DIMENSION statement
(2) COMMON statement
(3) EQUIVALENCE statement
(4) Type-statements
(5) IMPLICIT statement
(6) PARAMETER statement
(7) EXTERNAL statement
(8) INTRINSIC statement
(9) SAVE statement
(10) STACK statement

All specification statements are nonexecutable.

5.8.1 DIMENSION Statement

A DIMENSION statement is used to declare the symbolic names and dimension specifications of arrays.

5.8.1b
A DIMENSION statement is of the form:

DIMENSION a (d) [,a(d)] ...

where each a(d) is an array declarator.

5.8.1c
Each declarator name a appearing in a DIMENSION statement declares a to be an araay in that program unit. Note that array declarators may also appear in COMMON statements and type-statements. Only one appearance of a symbolic name as an array declarator name in a program unit is permitted.

5.8.2 COMMON Statement

A COMMON statement is of the form

COMMON / cb / nlist / cb / nlist ...

where: cb is a common block name

nlist is a list of the form a [, a] ...

a is either a variable name, an array name, or an array declarator. Dummy arguments are not permitted.

5.8.2c
Each omitted cb specifies the blank common block. If the first

cb is omitted, the first two slashes are optional.

5.8.2d
In each COMMON statement, the entities occurring in an nlist following a block name cb are declared to be in common block cb. 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 common. Alternatively, the appearance of two slashes with no block name between them declares the entities that follow to be in blank common.

5.8.2e
Any common block name cb or an omitted cb for blank common may occur more than once in one or more COMMON statements in a program unit. The list nlist following each successive appearance of the same common block name is treated as a continuation of the list for that common block name.

5.8.2f
If a character variable or character array is in a common block, all of the entities in that common block must be of type character. Note that entities associated with an entity in a common block are considered to be in that common block.

5.8.2g
Within a program unit, variables and arrays in each common block have consecutive storage in the order of their appearance in the lists in one or more COMMON statements. An arrayname or array declarator causes all of the elements of that array to have consecutive storage in the order specified in 5.2.4. Each real, integer, or logical variable or array element has one storage unit, each double precision or complex variable or array element has two consecutive storage units, and each character variable or array element has one character storage unit for each unit of length as specified in a CHARACTER statement (8.5.2). Within a program unit, each common block will have unique storage. An entiry appearing in a list nlist may not be associated with any other entiry appearing in any list nlist of any COMMON statement within that program unit

5.8.2h
The size of a common block is measured in storage units or character storage units and is the sum of the storage of variables and arrays declared in COMMON statements to be in the block plus any increase in size of the block caused by the use of EQUIVALENCE statements. EQUIVALENCE statements do not affect the order of assignment of storage to entities in common blocks, but their use may increase the size of a block

5.8.2i
Within an executable program, all named common blocks that have the same name must be of the same size and will have the same storage. All blank common blocks within an executable

program are not required to be of the same size, but all such blocks begin at the same storage unit or character storage unit and thus share some of the same storage. Any entities that share the same storage become associated and become defined and undefined according to the rules in 5.17.3 and 5.17.4.

5.8.2.1 Differences between Named Common and Blank Common - A blank common block has the same properties as a named common block, except for the following:

(1) Entities in named common blocks may be initially defined by means of a DATA statement in a block data subprogram, but entities in blank common may not be initially defined (Section 9).

(2) In different program units of an executable program, a specific named common block must be of the same length in all program units in which it appears, but blank common blocks may be of different lengths.

(3) Execution of a RETURN or END statement sometimes causes undefinition of entities in named common blocks, but never causes undefinition of entities in blank common (15.11).

5.8.3 EQUIVALENCE Statement

The EQUIVALENCE statement is used to specify the sharing of storage by two or more entities.

5.8.3b
An EQUIVALENCE statement is of the form:

EQUIVALENCE (*nlist*) [,(*nlist*)] ...

where: *nlist* is a list of the form *a*, *a* [,*a*]...

a is either a variable name, an array element name, an array name, or a character substring name. Each list must contain at least two names. Dummy argument names may not appear in the list.

If *a* is an array element or character substring name, its subscript expressions or substring expressions must be integer constant expressions.

5.8.3c
Each of the entities in a given list *nlist* shares some or all of the same storage. If a two-storage-unit entity and a one-storage-unit entity both appear in the same list *nlist*, the latter will share storage with the first storage unit of the former. The EQUIVALENCE statement should not be used to equate mathematically two or more entities. There is no

implied type conversion when the equivalenced entities are of different type. If a variable and an array both appear in the same list nlist, the variable does not have array properties and the array does not have the properties of a variable.

5.8.3d
An entity of type character may be equivalenced only with other entities of type character. The lengths of the equivalenced entities are not required to be the same. The first character of the equivalenced entities share storage and therefore are associated. Any adjacent characters in equivalenced character entities also share storage and are associated. For example, given:

```
CHARACTER A*4, B*4  C*3(2)
EQUIVALENCE (A,C(1)), (B,C(2))
```

then the association of A, B, and C can ge graphically illustrated as:

```
01 02 03 04 05 06 07

 ----A-----
       ------B-----
 --C(1)-- --(C(2)--
```

5.8.3.1 Array Names and Array Element Names in EQUIVALENCE Statements - If an array element name appears in an EQUIVALENCE statement, the number of subscript expressions must be either one or n, where n is the number of dimensions in the array declarator for the array name. If the array subscript contains only one expression, the subscript value is computed according to Table 1 (5.4.3) for a one-dimensional array.

5.8.3.1b
The use of an array name unqualified by a subscript in an EQUIVALENCE statement has the same effect as using an array element name that identifies the first element of the array.

5.8.3.2 COMMON and EQUIVALENCE - Storage of variables and arrays declared explicitly in a common block must be consecutive in the order specified in the COMMON statement. Within a program unit, storage is unique for each entity appearing in a list in a COMMON statement unless EQUIVALENCE statements cause it to be shared with entities not appearing in a list in a COMMON statement. An entity equivalenced with an entity in a COMMON statement is considered to be in that common block.

5.8.3.2b
When two or more variables or array elements share storage because of the effects of EQUIVALENCE statements, at most one of those variables or arrays may appear in the lists of the COMMON statements within a program unit.

5.8.3.2c
An EQUIVALENCE statement does not alter the ordering of entities in a common block, but it may lengthen a common block; the only such lengthening permitted is that which extends a common block beyond the last entity for that block declared directly by any COMMON statement in the same program unit.

5.8.3.2d
Information contained in Sections 5.5.2.4, 5.5.5.1 and this 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, or to cause consecutive array elements to have storage units that are not consecutive.

5.8.4 Type-Statements

A type-statement is used to override or confirm implicit typing and may specify dimension information.

5.8.4b
The appearance of a symbolic name in a type-statement specifies the data type for that name for all appearances in the program unit.

5.8.4c
Subroutine names, common block names, program names, and block data subprogram names cannot appear in a type-statement. Symbolic names of constants, which appear in PARAMETER statements, may appear in type-statements only in dimension bound expressions and character length specifications.

5.8.4d
If a generic function name (Table 5) appears in a type-statement within a program unit, that name loses its generic property in that program unit.

5.8.4.1 INTEGER, INTEGER *2, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, BIT, and ADDRESS TYPE-statements - INTEGER, INTEGER *2, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, BIT, or ADDRESS type statements are of the form:

typ v [,v] .

where: typ is INTEGER, INTEGER *2, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, or BIT, or ADDRESS

v is a variable name, an array name, a function or function name, or an array declarator.

5.8.4.2 CHARACTER Type-Statements - A CHARACTER type-statement is of the form:

CHARACTER *len [,] nam [,nam] ...

where: nam is of one of the forms:

v *len

a *len (d)

v is a variable name or a function name

a is an array name

a(d) is an array declarator

len is the length (number of characters) of a character variable, character array element, or character function.

5.8.4.2b
len is one of the following:

(1) An unsigned, nonzero, integer constant, except that the symbolic name of a constant is not permitted unless it is enclosed in parentheses.

(2) An arithmetic expression enclosed in parentheses in which all constants and variables are integer entities. Variables must be dummy arguments or in a common block. Function and array element references must not appear in the expression.

5.8.4.2c
A length len immediately following the word CHARACTER is the length specification for each entity in that statement not having its own length specification. A length specification immediately following an entity is the length specification for only that entity. Note that for an array, the length specified is for each array element. If a length is not specified for an entity, its length is one.

5.8.4.2d
An entity declared in a CHARACTER statement must have a length specification that is a constant expression unless that entity is a dummy argument and is a variable or an array. When an entity has a len of (*) declared, that entity must be a dummy argument, and the dummy argument assumes the length of the associated actual argument. Note that lengths of associated actual arguments are permitted to be different for different executions of the referenced subprogram.

5.8.4.2e
The length specified for a character function must be the same as the length of the character result of the function.

5.8.5 IMPLICIT Statement

An IMPLICIT statement is used to change or confirm the default implicit integer and real typing.

5.8.5b
An IMPLICIT statement is of the form:

IMPLICIT *typ* (*a* [,*a*]...) [,*typ* (*a* [,*a*]...)]...

where: *typ* is INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, CHARACTER **len* , INTEGER *2, BIT, or ADDRESS

a is either a range of letters in alphabetical order or a single letter. A range is denoted by the first and last letters of the range separated by a minus sign. Writing a range of letters a_1 - a_2 has the same effect as writing the single letters a_1 through a_2, inclusive, in alphabetical order.

len is the length of the character entities and is an unsigned, nonzero, integer constant. If *len* is not specified, the length has an implied value of one.

5.8.5c
An IMPLICIT statement specifies a type for all variables, arrays, function (except predefined functions), and function entries that begin with any letter that appears in a range of letters. IMPLICIT statements do not change the type of any intrinsic functions listed in Table 34.

5.8.5d
Type specification by an IMPLICIT statement may be overridden for any particular variable, array, function, or function entry name by the appearance of that name in a type-statement. An explicit type specification in a FUNCTION statement overrides an IMPLICIT statement for the name of that function subprogram.

5.8.5e
Within the specification statements of a program unit, IMPLICIT statements must precede all other specification statements except PARAMETER statements. A program unit may contain any number of IMPLICIT statements.

5.8.5f
The same letter may not appear or be included in a range of letters more than once in the group of IMPLICIT statements in any program unit.

5.8.6 PARAMETER Statement

A PARAMETER statement allows a constant to be given a symbolic name.

5.8.6b
A PARAMETER statement is of the form:

PARAMETER n=c [,n=c]...

where: n is a symbolic name

c is a constant other than a Hollerith constant.

5.8.6c
Each symbolic name n is the name of a constant and becomes defined to the value of the constant c that appears on the right of the equals. Once such a symbolic name is defined, that name may appear in that program unit in any subsequent statements, except PARAMETER and FORMAT statements, whereever a constant may appear and the effect is the same as if the constant appeared there instead of the name. Such a name may not appear in any position that does not allow a constant to appear. The form of the constant in the PARAMETER statement must be a form that is allowed to appear at the position where the name of the constant appears. The symbolic name of a constant may not appear as part of another constant, except that the symbolic name of a real constant may appear as either the real or the imaginary part of a complex constant.

5.8.6d
A symbolic name in a PARAMETER statement may identify only its corresponding constant in that program unit. Such a name may not appear more than once in PARAMETER statements within the same program unit and therefore may not be used as a constant in a PARAMETER statement.

5.8.6e
The symbolic name of a constant assumes the type implied by the form of its corresponding constant. The initial letter of the name has no effect on its type. The symbolic name of a character constant assumes the length attribute of its character constant. Symbolic names of constants may not appear in type-statements except in dimension bound expressions within array declarators and in character length specifications.

5.8.7 EXTERNAL Statement

An EXTERNAL statement is used to identify symbolic names as representing external procedures.

5.8.7b
An EXTERNAL statement is of the form:

EXTERNAL v [,v]...

where: each v is an external procedure name or entry name.

5.8.7c
Appearance of a name in an EXTERNAL statement declars that name to be an external procedure name. If an external procedure name or an entry name is an actual argument, it must appear in an EXTERNAL statement in that program unit.

5.8.7d
If an intrinsic function name (Table 3) or a statement function name appears in an EXTERNAL statement in a program unit, that name becomes the name of some external procedure and that intrinsic function or statement function is not available for reference in that program unit.

5.8.7e
Only one appearance of a symbolic name in all of the EXTERNAL statements of a program unit is permitted.

5.8.8 INTRINSIC Statement

An INTRINSIC statement is used to permit intrinsic functions as actual arguments.

5.8.8b
An intrinsic statement is of the form:

INTRINSIC v [,v]...

where each v is an intrinsic function name (Table 34).

5.8.8c
If an intrinsic function is an actual argument, it must appear in an INTRINSIC statement in that program unit. The intrinsic functions for choosing the largest or smallest value may not appear in an INTRINSIC statement. Note that a symbolic name may not validly appear in both an EXTERNAL and an INTRINSIC statement in any program unit.

5.8.8d
If a generic function name (Table 5) appears in an INTRINSIC statement in a program unit, that name must also be the name of a specific intrinsic function which may then be used as an actual argument. The appearance of a generic function name in an INTRINSIC statement does not cause that name to lose its generic property.

5.8.8e
Only one appearance of a symbolic name in all of the INTRINSIC statements of a program unit is permitted.

5.8.9 SAVE Statement

A SAVE statement is used to cause entities and named common blocks to remain defined following the execution of a RETURN or END statement in that program unit.

5.8.9b
A SAVE statement is of the form:

SAVE a ,a ...

where a is a named common block name preceded and followed by a slash, a variable name, or an array name. Redundant appearances of an item are permitted.

5.8.9c
Dummy arguments, procedure names, and entities in common may not appear in a SAVE statement.

5.8.9d
A SAVE statement without a list is treated as though it contained the names of all allowable items in that program unit.

5.8.9e
The appearance of a common block name in a SAVE statement has the effect of specifying all the entities in that common block. No entities appearing in a SAVE statement become undefined as the result of the execution of a RETURN or END statement in the program unit where the SAVE statement appears.

5.8.10 STACK Statement

A STACK statement is used to declare a push down stack. It gives the stack its type and maximum number of elements that it can hold. The form of the STACK statement is:

typ STACK*n nam [,nam] ...

where typ is:

INTEGER,INTEGER*2,REAL,DOUBLE PRECISION,COMPLEX,ADDRESS,
LOGICAL,CHARACTER [*len],or BIT

n is the maximum number of elements the stack will hold

nam is the stack variable being declared.

5.8.10b
n is one of the following:

(1) An unsigned, nonzero, integer constant, except that the symbolic name of a constant is not permitted unless it is enclosed in parentheses.

(2) An arithmetic expression enclosed in parentheses in which all constants and variables are integer entities. Variables must be dummy arguments or in a common block. Function and array element references must not appear in the expression.

(3) An asterisk in parentheses, (*) which means that the stack name(s) are dummy arguments and they assume the number of elements of the associated actual arguments.

5.8.10.1
Stacks that have not been declared as dummy arguments may appear in EQUIVALENCE and COMMON statements. Stacks which are declared in EQUIVALENCE statements should have the same number of elements. Character stacks can only be EQUIVALENCE'd to other character stacks.

5.9 DATA STATEMENT

A DATA statement is used to provide initial values for variables, array, and array elements. A DATA statement is nonexecutable and may appear in a program unit anywhere after the specification statements (if any).

5.9b
A DATA statement is of the form:

DATA nlist / clist/ , nlist / clist/ ...

where: nlist is of the form

a [,a] ...

where: a is the name of a variable, array, array element, or an implied-DO list.

clist is of the form:

con [,con] ...

where: con is either a constant or a constant pre-

ceded by r*. When the form r* appears before a constant, it indicates r successive appearances of the constant.

r is a nonzero, unsigned, integer constant.

5.9c
Dummy arguments, function and function entry names, character substring names, and entities in blank common (including entities associated with an entity in blank common) may not appear in the list nlist. Entities in a named common block may appear in the list nlist within a block data subprogram only.

5.9d
There must be the same number of items specified by the list nlist and the list clist. There is a one-to-one correspondence between the nlist items and the constants specified by clist such that the first item of nlist corresponds to the first constant of clist, etc. By this correspondence, the initial value is established and the entity is initially defined. In an array name without a subscript is in the list, there must be one constant for each element of that array. The ordering of array elements is sequential according to array element subscript value.

5.9e
The type of the nlist entity and the type of the corresponding clist constant must agree when either is of type character, complex, or logical. When the nlist entity is of type integer, real, or double precision, the corresponding clist constant must also be of the type integer, real, or double precision; if necessary, the clist constant is converted to the type of the nlist entity according to the rules for arithmetic assignment statements. Note, however, that when the clist constant is a Hollerith constant, the nlist entity may be of any type except character.

5.9f
Any variable or array element may be initially defined except for:

(1) an entity that is a dummy argument, an entity in blank common, or an entity associated with an entity in blank common, or

(2) a variable in a function subprogram whose name is also the name of that function or an entry in that function.

5.9g
An entity may not be initially defined more than once in an executable program. Only one entity of a set of associated entities may be initially defined explicitly in the same executable program. Note that in an executable program, a

storage unit or character storage unit may not have its value initially defined more than once.

5.9h
All initially defined entities are defined when an executable program begins execution. All entities not initially defined are undefined at the beginning of execution of an executable program.

5.9.1 Implied-DO in DATA Statement - An implied-DO list in a DATA statement is of the form:

(nlist, $i = m_1, m_2, m_3$)

where: nlist is a list containing array elements and implied-DO lists

i is the name of an integer variable, called the implied-DO control variable

m_1, m_2, m_3 are either constants or the control variable of an implied-DO list that has this implied-DO list within its range.

5.9.1b
The range of an implied-DO list is the immediately preceding list nlist. An interation count is calculated from m_1, m_2, and m_3 exactly as the DO-loop. The iteration count must be positive. The appearance of a control variable name in a DATA statement does not cause definition or undefinition of a variable of the same name in the same program unit.

5.9.1c
Any subscript expression and implied-DO parameter must consist of integer constants and implied-DO control variables only. The value of any control variable appearing within a subscript expression or as an implied-DO parameter in an nlist must have a constant value specified within that list. Subscript expressions must not contain exponentiation or function references.

5.9.2 Character Constants in DATA Statements

An entity in the list nlist that corresponds to a character constant must be of type character. If the length of the character entity in the list nlist is greater than the length of its corresponding character constant, the additional rightmost characters in the entity are initialized with blank characters. If the length of the character entity in the list nlist is less than the length of its corresponding character constant, the additional rightmost characters in the constant are ignored. Note that initialization of a character entity causes definition of all of the characters in that entity, and that each character constant does not initialize more than one variable or array element.

5.9.3 Hollerith Constants in DATA Statements

A Hollerith constant may appear in the list clist, and the corresponding entity in the list nlist may be of any type except complex or character.

5.9.3b
If the entity is of type integer, real, or logical, then the number of characters n in the corresponding Hollerith constant must be less than or equal to 4. If n is less than 4, the entity is initialized with the n Hollerith characters extended on the right with 4-n blank characters. If the entity is of type double precision, then n must be less than or equal to 8. If n is less than 8, the double precision entity is initialized with the n Hollerith characters extended on the right with 8-n blank characters.

5.9.3c
Note that each Hollerith constant does not initialize more than one variable or array element.

5.10 Assignment Statements

There are five kinds of assignment statements

(1) Arithmetic assignment statement

(2) Logical assignment statement

(3) Statement label assignment (ASSIGN) statement

(4) Character assignment statement

(5) Address assignment statement

Completion of execution of an arithmetic, a logical, a character, or an address assignment statement causes definition of an entity.

5.10.1 Arithmetic Assignment Statement

An arithmetic assignment statement is of the form:

$$v = e$$

where: v is a variable name or an array element name of type integer, real, double precision, or complex.

e is an arithmetic expression.

5.10.1b
Execution of this statement causes the evaluation of the expression e and the assignment and definition of v as established by the rules in Table 2.

5.10.1c

Table 2

Rules for Arithmetic Assignment and Conversion of e to v

Type of v	Type of e	Assignment Rule
Integer	Integer	Assign
	Real	Fix and assign
	Double	Fix and assign
	complex	Fix real part and assign
Real	Integer	Float and assign
	Real	Assign
	Double	Real assign
	Complex	Assign real part
Double	Integer	DP float and assign
	Real	DP float and assign
	Double	Assign
	Complex	DP float real part and assign
Complex	Integer	Float and assign real part; imaginary part = 0
	Real	Assign real part; imaginary part = 0
	Double	Real assign real part; imaginary part = 0
	Complex	Assign

5.10.1c
Notes for Table 2:

(1) Double means double precision.

(2) Assign means transmit the resulting value, without change, to the entity.

(3) Real assign means transmit to the entity as much precision of the most significant part of the resulting value as real datum can contain.

(4) Fix means truncate any fractional part of the result and convert that value to an integer datum.

(5) Float means convert the value to a real datum.

(6) D P float means convert the value to a double precision datum, retaining in the process as much of the precision of the value as a double precision datum can contain.

5.10.2 Logical Assignment Statement

A logical assignment statement is of the form:

v = e

where: v is the name of a logical variable or logical array element

e is a logical expression

5.10.2b
Execution of this statement causes the evaluation of the logical expression e and the assignment and definition of v with the value of e. Note that e must have a value of either true or false.

5.10.3 Statement Label Assignment (ASSIGN) Statement

A statement label assignment statement is of the form:

ASSIGN s TO i

where: s is a statement label

i is an integer variable name

5.10.3b
Execution of this statement causes the statement label to be assigned to the integer variable. The statement label must be the label of a statement in the same program unit in which the ASSIGN statement appears. The statement label must be the label of an executable statement or a FORMAT statement. If the label of an executable statement has been assigned to an integer variable, subsequent execution in the same program unit of any assigned GO TO statement that contains the same integer variable will cause the statement identified by the assigned statement label to be executed next, provided there has been no intervening definition or undefinition of the variable.

5.10.3c
If the label of a FORMAT statement has been assigned to an integer variable, subsequent execution in the same program unit of any formatted input/output statement that contains the same integer variable as a format identifier will use the FORMAT statement identified by the assigned statement label, provided there has been no intervening definition or undefinition of the variable.

5.10.3d
After an integer variable has been assigned a statement label, that integer variable may not be referenced in any statement other than an assigned GO TO statement or as a

format identifier in an input/output statement until it has been defined to an integer value. It may be subsequently assigned to the same or a different statement label.

5.10.3e
Completion of execution of an ASSIGN statement causes undefinition of the integer variable as an integer. Any subsequent definition of the variable as an integer removes the previous assignment to a statement label.

5.10.4 Character Assignment Statement

A character assignment statement is of the for-:

v = e

where: v is the name of a character variable, character array element, or character substring

e is an expression of type character

5.10.4b
Execution of this statement causes the evaluation of the expression e and the assignment and definition of v with the value of e. The character positions referenced in e may not be the character positions being defined in v. All of the character positions referenced in e must be defined. v and e may have different length attributes. If the length of v is greater than the length of e, the effect is as though e is extended to the right with blank characters until it is the same length as v and then assigned. If the length of v is less than the length of e, the effect is as though e is truncated from the right until it is the same length as v and then assigned.

5.10.4c
If v is a substring, only the character positions specified are defined. The status of character positions not specified by the substring is unaffected.

5.10.5 Address Assignment Statements

An address assignment statement is of the form:

v = e

where v is the name of an address variable or address array element

e is an expression of type address

5.10.5b
An address expression is one that contains only address constants, address mode variables, integer constants, and integer variables.

Execution of an address assignment statement causes the evaluation of the address expression e and the assignment and definition of v with the value of e.

5.10.5c
Associated with an address variable is an inverse value operation @ which gives the value that the address variable points to. For example,

```
ADDRESS A
A = A'I'
J = @A
```

J will be assigned the value of the argument whose address is in A, in this case the value of I.

5.11 CONTROL STATEMENTS

Control statements may be used to control the execution sequence within a program unit.

5.11b
There are ten control statements:

(1) Unconditional GO TO Statement

(2) Assigned GO TO Statement

(3) Computed GO TO Statement

(4) Arithmetic IF Statement

(5) Logical IF Statement

(6) DO Statement

(7) CONTINUE Statement

(8) STOP Statement

(9) PAUSE Statement

(10) END Statement

5.11.1 Unconditional GO TO Statement

An unconditional GO TO statement is of the form:

```
GO TO s
```

where s is a statement label of an executable statement within the same program unit in which the GO TO statement appears.

5.11.1b
Execution of this statement causes the statement identified by the statement label to be executed next.

5.11.2 Assigned GO TO Statement

An assigned GO TO statement is one of the forms:

```
GO TO i
GO TO i [,] ( s [,s]... )
```

where: *i* is an integer variable name

s is a statement label of an executable statement within the same program unit in which the GO TO statement appears. The same statement label may appear more than once in the GO TO statement.

5.11.2b
At the time of execution of an assigned GO TO statement, the current value of *i* must have been assigned by the prior execution of an ASSIGN statement to a statement label of an executable statement; the execution of the assigned GO TO statement causes the statement identified by that statement label to be executed next. The last definition of the variable in an assigned GO TO statement must have occurred in the same program unit as the assigned GO TO statement. If the form with the parenthesized list is used, then the statement label assigned to *i* must be one of the statement labels in the list.

5.11.3 Computed GO TO Statement

A computed GO TO statement is of the form:

```
GO TO (s [,s]... ) [,] i
```

where: *i* is an integer expression

s is a statement label of an executable statement within the same program unit in which the GO TO statement appears. The same statement label may appear more than once in the GO TO statement.

5.11.3b
Execution of this statement causes the statement identified by the jth statement label to be executed next, where j is the integer value of *i* at the time of the execution and $1 \le j \le n$, where n is the number of statement labels within the parentheses. If $j < 1$ or $j > n$, then execution proceeds as though this statement were a CONTINUE statement.

5.11.4 Arithmetic IF Statement

An arithmetic IF statement is of the form:

```
IF (e) s1, s2, s3
```

where: *e* is any arithmetic expression of type integer, real, or double precision

s_1 ,s_2 and s_3 are statement labels of executable statements within the same program unit. The same statement label may appear more than once.

5.11.4b
The arithmetic IF statement is a three-way branch. Execution of this statement causes evaluation of the expression e following which the statement identified by the statement label s_1, s_2, or s_3 is executed next as the value of e is less than zero, equal to zero, or greater than zero, respectively.

5.11.5 Logical IF Statement

A logical IF statement is of the form:

IF(e) st

where: e is a logical expression

st 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 st is executed as though it were a CONTINUE statement. If the value of e is true, statement st is executed.

5.11.6 DO Statement

A DO statement is used to define a loop, called a DO-loop.

5.11.6b
A DO statement is of the form:

DO s [,] i = m_1, m_2, [,m_3]

where: s is the statement label of an executable statement. This statement, called the terminal statement of the DO-loop, must physically follow and be in the same program unit as the DO statement.

i ,called the control variable, is an integer, real, or double precision variable name.

m_1 is called the initial parameter; m_2 is called the terminal parameter; and m_3 is called the incrementation parameter. m_1, m_2, and m_3 are each an integer, real, or double precision expression. If m_3 does not explicitly appear, a value of one is implied for the imcrementation parameter.

5.11.6c
The terminal statement of a DO-loop may not be an unconditional GO TO, assigned GO TO, arithmetic IF, RETURN, STOP, END, or DO statement. If the terminal statement of a DO-loop is a logical IF, it may contain any executable statement except a DO statement, another logical IF statement, or END statement.

5.11.6.1 Range of a DO-loop

The range of a DO-loop is the set of executable statements from and including the first executable statement following the DO statement that defines the DO-loop, to and including the terminal statement of the DO-loop.

5.11.6.1b
If a DO statement appears within the range of a DO-loop, the range of the DO-loop defined by that DO statement must be within the range of the outer DO-loop. Note that the definition of range of a DO-loop permits both DO-loops to share a terminal statement.

5.11.6.2 Active and Inactive Do-loops

A DO-loop is either active or inactive. Initially inactive, it becomes active only when the execution of its DO statement defines a nonzero iteration count (5.11.6.3).

5.11.6.2b
Once active, the DO-loop becomes inactive only

(1) when its iteration control (5.11.6.4) decrements its iteration count to zero,

(2) when the execution of a RETURN statement in its program unit or a STOP statement in its executable program occurs,

(3) when it is in the range of a DO-loop that becomes inactive, or

(4) when it is in the range of a DO-loop whose DO statement is executed.

5.11.6.2c
When a DO-loop becomes inactive, its control variable retains its last defined value.

5.11.6.3 Executing a DO Statement

The effect of executing a DO statement is as follows:

(1) The DO statement parameters are evaluated. If necessary, they are converted to the type of the control variable according to the rules for arithmetic

assignment statements (Table 2). The value of $\underline{m}_3$ must not be zero. If conversion of $\underline{m}_3$ is necessary, its value after conversion must not be zero.

(2) The control variable is defined and has the value of the initial parameter.

(3) The iteration count is defined and is the largest integer value that does not exceed the value of the expression

$$(\underline{m}_2 - \underline{m}_1)/\underline{m}_3 + 1$$

If the value of the expression is less than zero, the iteration count is zero. Note that the iteration count is zero whenever:

$\underline{m}_1 > \underline{m}_2$ and $\underline{m}_3 > 0$, or
$\underline{m}_1 < \underline{m}_2$ and $\underline{m}_3 < 0$.

(4) If the iteration count is not zero, then the DO-loop becomes active. This is followed by the normal execution of the statements in the range of the DO-loop until the terminal statement is reached. Except by the iteration control described in 5.11.6.4, the control variable of this DO-loop may not be defined in the range of the DO-loop.

5.11.6.3b
If the iteration count is zero, then the DO-loop becomes inactive and execution continues with the processing of the terminal statement as described below.

5.11.6.4 Processing the Terminal Statement of a DO-loop

The processing of the terminal statement of a DO-loop is as follows:

(1) If any DO-loop that has this statement as its terminal statement is inactive, the terminal statement is executed as though it were a CONTINUE statement. If execution of the terminal statement results in a transfer of control, then no additional action occurs. Otherwise, the iteration control processing follows.

(2) If any DO-loop that has this statement as its terminal statement is active, the iteration control of the active DO-loop whose DO statement has this terminal statement and was most recently executed is selected for processing. Iteration control begins with the algebraic addition of the value of $\underline{m}_3$ defined by execution of the DO statement. The iteration count is decremented by one. If this DO-loop is still active, normal execution of the statements in the range of the DO-loop follows.

assignment statements (Table 2). The value of $\underline{m}_3$ must not be zero. If conversion of $\underline{m}_3$ is necessary, its value after conversion must not be zero.

(2) The control variable is defined and has the value of the initial parameter.

(3) The iteration count is defined and is the largest integer value that does not exceed the value of the expression

$$(\underline{m}_2 - \underline{m}_1)/\underline{m}_3 + 1$$

If the value of the expression is less than zero, the iteration count is zero. Note that the iteration count is zero whenever:

$\underline{m}_1 > \underline{m}_2$ and $\underline{m}_3 > 0$, or
$\underline{m}_1 < \underline{m}_2$ and $\underline{m}_3 < 0$.

(4) If the iteration count is not zero, then the DO-loop becomes active. This is followed by the normal execution of the statements in the range of the DO-loop until the terminal statement is reached. Except by the iteration control described in 5.11.6.4, the control variable of this DO-loop may not be defined in the range of the DO-loop.

5.11.6.3b
If the iteration count is zero, then the DO-loop becomes inactive and execution continues with the processing of the terminal statement as described below.

5.11.6.4 Processing the Terminal Statement of a DO-loop

The processing of the terminal statement of a DO-loop is as follows:

(1) If any DO-loop that has this statement as its terminal statement is inactive, the terminal statement is executed as though it were a CONTINUE statement. If execution of the terminal statement results in a transfer of control, then no additional action occurs. Otherwise, the iteration control processing follows.

(2) If any DO-loop that has this statement as its terminal statement is active, the iteration control of the active DO-loop whose DO statement has this terminal statement and was most recently executed is selected for processing. Iteration control begins with the algebraic addition of the value of $\underline{m}_3$ defined by execution of the DO statement. The iteration count is decremented by one. If this DO-loop is still active, normal execution of the statements in the range of the DO-loop follows.

(3) If the DO-loop was or becomes inactive, and all other DO-loops sharing this terminal statement are inactive, then control concludes. Normal execution of the statement following the terminal statement occurs next. Otherwise, iteration control continues by repeating step 2.

5.11.6.5 Transfer into the Range of a DO-loop

Transfer of control into the range of an inactive DO-loop is not permitted. Transfer of control into the range of an active DO-loop is permitted only if the DO-loop control variable has not been subsequently defined by means other than the incrementation control described in 5.11.6.4.

5.11.7 CONTINUE Statement

A CONTINUE statement is of the form:

```
CONTINUE
```

Execution of this statement causes continuation of the normal execution sequence.

5.11.8 STOP Statement

A STOP statement is of the form:

```
STOP [n]
```

where n is a decimal digit string of not more than five digits.

5.11.8b
Execution of this statement causes termination of execution of the executable program. At the time of termination, the digit string is accessible.

5.11.9 PAUSE Statement

A PAUSE statement is of the form:

```
PAUSE [n]
```

where n is a decimal digit string of not more than five digits or a character constant.

5.11.9b
Execution of this statement causes a cessation of execution of the executable program. Execution must be resumable. At the time of cessation of execution, the digit string or character string is accessible. Resumption of execution is not under control of the program. If execution is resumed without otherwise changing the state of the processor, the normal execution sequence is continued.

5.11.10 END Statement

An END statement physically terminates a program unit and may have the effect of a STOP or RETURN statement.

5.11.10b
An END statement is of the form:

```
END
```

5.11.10c
An END statement may not be labeled and it may not be continued. Columns 1 through 6 of the line on which the END statement is written must all contain blank characters.

5.11.10d
The END statement indicates to the processor the end of the written description of a program unit. Every program unit must physically terminate with an END statement.

5.11.10e
An END statement is executable. Execution of an END statement in a main program has the same effect as executing a STOP statement (5.11.8). Execution of an END statement in a subprogram has the same effect as executing a RETURN statement.

5.12 INPUT/OUTPUT STATEMENTS

Input statements provide the means of transferring data from external media to internal storage. This process is called reading. Output statements provide the means of transferring data from internal storage to external media. This process is called writing. Some statements cause conversion of the data.

5.12.b
In addition to the statements that transfer data, there are auxiliary input/output statements to manipulate the external medium, or to inquire about or describe the properties of the external medium.

5.12.c
In their general form, the input/output statements that transfer data contain three categories of information:

(1) Direction of transfer

(2) Control information that includes:

A reference to the source or destination of the data to be transferred.

Optional specification of conversion processes.

Optional directives for change of execution sequence

on the occurrence of certain events.

(3) A list specifying the data to be transferred.

An external medium is described in terms of records and files. It is referenced by means of a unit. No physical structures or specific devices are implied.

5.12.1 Records

A record is a basic concept. For example, a punched card is usually considered to be a record. However, a record does not necessarily correspond to a physical entity. There are four kinds of records:

(1) formatted

(2) unformatted

(3) free-field

(4) endfile

5.12.1.1 Formatted Records

A formatted record consists of a sequence of characters that are capable of representation in the processor. The length of a formatted record is measured in characters. Formatted records are read and written only by formatted input/output statements.

5.12.1.2 Unformatted Records

An unformatted record consists of a sequence of values in a processor-dependent form. An unformatted record may contain both character and noncharacter data. The length of an unformatted record is measured in storage units. The length L of an unformatted record in terms of storage units is given as:

$$L = n + \sum_i \left[\frac{k_i + g - 1}{g} \right]$$

where: n is the number of noncharacter storage units in the record

k_i is the number of characters in the i-th set of contiguous character items in the record

g is the maximum number of characters that can be stored in a single storage unit at one time

[ ] is the greatest-integer function.

Unformatted records are read and written only by unformatted input/output statements.

5.12.1.3 Free-field Records

A free-field record consists of a series of values formed as described in 5.12.10 from characters capable of representation in the processor. The length of a free-field record is measured in values; the number of values in the record is not necessarily known to the processor. Free-field records are read and written only by free-field input/output statements. The values which comprise a single free-field record are read by one or more successive free-field input statements and are written by one or more successive free-field output statements.

5.12.1.4 Endfile Records

An endfile record is written by an ENDFILE statement. An endfile record does not have a length property.

5.12.2 Files

A file is a set of zero, one, or more records. A record is a member of exactly one file. There are three kinds of files:

(1) sequential

(2) direct access

(3) storage

Sequential and direct access files are collectively referred to as external files.

5.12.2.1 File Existence

Any file that is connected at the beginning of execution of an executable program, or is capable of being connected after the beginning of execution, is said to exist for that program. A file may exist and contain no records; for example, a new file not yet written.

5.12.2.1b
To create a file means to cause a file to exist that did not previously exist. To delete a file means to terminate the existence of the file.

5.12.2.1c
Certain input/output statements, such as READ and WRITE, may refer only to files that exist. Other input/output statements such as INQUIRE, OPEN, and CLOSE, may refer to files that do not exist.

5.12.2.2 File Names

A file may have a name; a file that has a name is called a named file. The name of a named file is a character string. The set of allowable names is processor dependent. A processor need not permit named files.

5.12.2.3 Sequential Files

A nonempty sequential file has the following properties:

(1) Reading and writing of records is accomplished only by sequential input/output statements or free-field input/output statements.

(2) The records of the file exist as a totally ordered set. There is a first record and a last record. The order of the records is the order in which they were written.

(3) The records of the file may be either formatted, unformatted, free-field, or any combination thereof. The last record in the file may be an endfile record. Records may be of different lengths in the same file.

(4) The file has a position property, from which are derived the related concepts of next record, preceding record, current record, and initial point. Certain circumstances can cause the position of the file to become indeterminate.

(5) If the file is positioned between the ith record and the ith+1 record, the ith record is the preceding record and the ith+1 record is the next record and there is no current record.

(6) A current record exists only when the current position of the file is within a record rather than between two records. This condition is caused only by free-field input/output statements. If the current record is the ith record, then the ith+1 record, if it exists, is the next record. For i>1, the ith-1 record is the preceding record. Note that, if a file is positioned so that there is a current record, that record must be a free-field record.

(7) The initial point of the file is that unique point where the next record is the first record of the file. When the file is positioned at its initial point, there is no preceding record in the file and no current record.

(8) When the file is positioned after the last record, there is no current record and no next record in the file. The last record is the preceding record.

5.12.2.3b
A sequential file that contains no records is empty. An empty file has no preceding record, no current record, and no next record.

5.12.2.3.1 Sequences of Sequential Files

A sequence of sequential files has the following properties:

(1) The files exist as a totally ordered set. There is a first file and a last file. The order of the files is the order in which they were written.

(2) The files are separated by endfile records. An endfile record is part of the file that it terminates. The last file need not be terminated by an endfile record.

(3) The sequence has a position property, from which are derived the related concepts of next file, preceding file, current file, and initial point.

(4) If the sequence is positioned between the ith file and the ith+1 file, the ith file is the preceding file, the ith+1 file is the next file and there is no current file. This position is the initial point of the ith+1 file. The preceding record is the last record of the ith file.

(5) If the sequence is positioned between two records of a file, or within a record of a file, that file is the current file. If the current file is the ith file, then the ith+1 file, if it exists, is the next file. For i>1, the ith-1 file is the preceding file.

(6) If the sequence is positioned at the initial point of the first file, there is no preceding file, and no current file.

(7) When the sequence is positioned after the last file, there is no current file and no next file.

5.12.2.4 Direct Access Files

A direct access file has the following properties:

(1) Reading and writing of records is accomplished only by direct access input/output statements.

(2) The records of the file are either all formatted or all unformatted records. The file may not contain free-field records, endfile records, or a mixture of formatted and unformatted records.

(3) Every record in the file has the same length.

(4) Each record in the file is uniquely identified by a positive integer called the record number. The value of the record number of a record is established when the record is written. Once established, the record number of a record may never be changed, nor may a record be removed from the file.

(5) Records need not be read or written in the order of their record numbers. Any record may be written into the file while it is connected (5.12.3) to a unit, unless the file was opened with the READ ONLY specifier (5.12.11.1). Any record may be read from the file while it is connected to a unit, provided the record has been written subsequent to the creation of the file. For example, it is permissible to write record 3, even though records 1 and 2 have not been written.

(6) The file optionally may have a maximum-record-number property that is established when the file is created. Once established, the presence or absence of the maximum-record-number property may not be changed.

(7) If the file has the maximum-record-number property, the value of the maximum record number is established, and may only be established, when the file is created. No record in the file may have a record number that is greater than the maximum record number. If a file does not have the maximum-record-number property, then the records in the file may have any record number not exceeding some processor-dependent upper bound.

5.12.2.5 Storage Files

A storage file is an entity of type character. A record of a storage file is a character variable or a character array element. When used for input/output, these entities are called records and the collection of records is called a storage file.

5.12.2.5b
A storage file has the following properties:

(1) Reading and writing records is accomplished only by formatted sequential input/output statements.

(2) The file contains only formatted records.

(3) The file is a character variable, character array element, or character array. Note that a character substring is not permitted.

(4) If the file is a character variable or a character array element, it consists of a single record whose length is the same as the length of the variable or array element. If the file is a character array, it is treated as a sequence of character array elements. Each array element is a record in the file. The records exist as a totally ordered set and the ordering of the records in the file is the same as the ordering of the array elements in the array. Every record of the file has the same length, which is the length of an array element in the array.

(5) The variable or array element that is the record in the storage file becomes defined when that record is written.

(6) Only records that correspond to a variable or array element that is defined may be read by formatted sequential input statements.

(7) The character entity used as a record of a storage file may be defined (or undefined) by means other than a formatted sequential output statement. For example, the variable or array element may be defined by a character assignment statement.

5.12.3 Units

A unit is a means of referring to one of the following:

(1) a sequential file

(2) a direct access file

(3) a storage file

5.12.3b
A unit has a property of being connected or not connected. If connected, it refers to one of the above kinds of files. A unit may become connected by preconnection or by the execution of an OPEN statement. The property of connection is symmetric, i.e., if a unit is connected to a file, then the file is connected to the unit.

5.12.3c
Preconnection means that the unit is connected to a file at the beginning of execution of the executable program and therefore may be referenced by input/output statements without the prior execution of an OPEN statement.

5.12.3d
A unit must not be connected to more than one file at the same time and a file must not be connected to more than one unit at the same time. However, to change the status of a unit and to connect a unit to a different file, all input/output statements except OPEN, CLOSE, and INQUIRE must identify a unit that is connected to a file.

5.12.3.1 Kinds of Units

There are two kinds of units:

(1) External

(2) Internal

5.12.3b
An external unit is used to refer to a sequential file or a direct access file. An internal unit is used to refer to a storage file. An internal unit is always connected.

5.12.3.2 Unit Identifiers

In the following descriptions of input/output statements, u identifies a unit. The unit identifier u for an external unit is called an external unit identifier and is an integer, real, or double precision expression that has a zero or positive value when converted to an integer according to the rules for arithmetic assignment statements. One and only one unit is identified by each integer value within an executable program.

5.12.3.2b
The unit identifier for a storage file is called an internal unit identifier and is the symbolic name of a character variable, a character array, or a character array element. Note that a character substring is not permitted.

5.12.4 Format Identifiers

In the following descriptions of input/output statements, f identifies a format specification (Section 5.13). The format identifier f may be any one of the following:

(1) The statement label of a FORMAT statement in the same program unit as the input/output statement.

(2) An integer variable name that has been assigned the statement label of a FORMAT statement in the same program unit as the input/output statement.

(3) A character expression or character array name (5.13.1.1). Note that this includes a character constant whose value is a format specification.

(4) An array name not of character type (5.13.1.3).

5.12.5 Control Information Specifiers

The control information given in each input/output statement consists of a list of positional or keyword control information specifiers. Except for the positional

forms of the unit and format specifiers, all specifiers are keyword specifiers and may appear in any order. Note that a list that includes keyword specifiers is always enclosed in parentheses.

5.12.5.1 Unit and Format Specifiers

In the following descriptions of input/output statements, unt identifies a unit specifier. unt is of one of the forms:

u (positional)

UNIT=u (keyword)

Where u is a unit identifier.

5.12.5.1b
If u is an external unit identifier, unt is referred to as an external unit specifier. If u is an internal unit identifier, unt is referred to as an internal unit specifier.

5.12.5.1c
A format specifier is identified by fmt. fmt is of one of the forms:

f (positional)

FMT=f (keyword)

where f is a format identifier.

5.12.5.1.1 Use of Positional Form

If the positional form u of the unit specifier appears in an input/output statement, it must appear as the first specifier in that statement. If the positional forms of both the unit and format specifiers appear in the same input/output statement, the positional form u must appear as the first specifier and the positional form f must appear as the second specifier.

5.12.5.1.2 Use of Keyword Form

If both the unit and format specifiers appear in an input/output statement and the keyword form UNIT=u is used, then the keyword form FMT=f must also be used. In this case, the unit and format specifiers may appear in any order among the other keyword specifiers.

5.12.5.2 Error Specifier

An error specifier is of the form:

ERR = s

where s is the label of an executable statement that appears in the same program unit as the error specifier.

5.12.5.2b
If an input/output statement contains an error specifier and if the processor encounters an error condition during the execution of that statement, then the execution of that input/output statement terminates and execution continues with the statement labeled s. The position of the unit specified in the input/output statement, if it is connected to a sequential file, becomes indeterminate. If the error condition occurs during execution of an input statement, the entities specified by the input list become undefined. If an input statement defines the value of a character or Hollerith format field descriptor, that value becomes undefined.

5.12.5.2c
If an error condition occurs during execution of an input/output statement that does not contain an error specifier, execution of the executable program is terminated.

5.12.5.3 End-of-file Specifier

An end-of-file specifier is of the form:

END = s

where s is the label of an executable statement that appears in the same program unit as the end-of-file specifier.

5.12.5.3b
If an input statement contains an end-of-file specifier and if the processor encounters an end-of-file condition during the execution of that statement, then execution of that input statement terminates, the entities specified by the input list become undefined, and execution continues with the statement labeled s.

5.12.5.3c
Execution of an executable program is terminated when an end-of-file condition is encountered during execution of an input statement that does not contain an end-of-file specifier.

5.12.5.3d
An end-of-file condition exists if either of the following events occurs:

(1) An endfile record is encountered during the reading of sequential formatted records, sequential unformatted records, or sequential free-field records. In this case, the file referenced by the input statement is positioned so that the endfile record is the preceding record before execution continues with the statement labeled s.

(2) An attempt is made to read a record beyond the end of a storage file.

5.12.6 Input/Output Lists

An input/output list specifies the entities whose values are transferred by an input/output statement.

5.12.6b
An input/output list is a list of list items or implied-DO lists.

5.12.6.1 Input List Items

In an input statement, a list item must be one of the following:

(1) a variable name

(2) an array element name

(3) a character substring name

(4) an array name

(5) an array block item

5.12.6.2 Output

In an output statement, a list item must be one of the following:

(1) an expression other than a Hollerith constant

(2) an array name

(3) an array block item

5.12.6.3

When an array name appears as a list item, it is treated as if all of the elements of the array were specified in the order given by array element ordering.

5.12.6.1.1 Array Block Items

An array block item is of one of the forms:

$$a_1 : a_2$$

$$a_1 :$$

$$: a_2$$

where a_1 and a_2 are array element names within the same array. The subscript value of a_1 must not exceed the subscript value of a_2.

5.12.6.1.1b
When an array block item appears as a list item, it is treated as if all of the elements of the array beginning with a_1 and ending with a_2 were specified in the order given by array element ordering. If a_1 is omitted, the sequence begins with the first element of the array named in a_2. If a_2 is omitted, the sequence ends with the last element of the array named in a_1. During the execution of an input/output statement, the subscript values of both a_1 and a_2 are computed by the processor before any transmission of values occurs between the array block and the input/output unit.

5.12.6.2 Implied-DO Lists

An implied-DO list is of the form:

(dlist, i = m_1, m_2 [, m_3])

where: dlist is an input/output list

i, m_1, m_2, and m_3 are as specified for the DO statement.

5.12.6.2b

The range of an implied-DO list is the set of items in the list dlist. Note that dlist may contain implied-DO lists. The iteration count and the values of the control variable i are established from m_1, m_2, and m_3 exactly as for a DO-loop. In input statements, the control variable i may appear within dlist only in subscript expressions. When an implied-DO list appears in an input/output list, it is treated as if dlist were specified once for each iteration of the implied-DO list.

5.12.7 Execution of Input/Output Statements

The entities specified by the input/output list are transferred in the order of the list items.

5.12.7.1 Execution of Input Statements

Execution of an input statement causes values to be transferred from the input medium to the entities specified by the input list. As a value is transferred to an entity, that entity becomes defined. Note that this may affect subsequent list items. An input statement must not specify the reading of more data from a record than the record contains.

5.12.7.2 Execution of Output Statements

Execution of an output statement causes values of the entities specified by the output list to be transferred to the output medium. Every entity whose value is to be transferred must be defined.

5.12.8 Unformatted Input/Output Statements

5.12.8.1 Unformatted Input Statements

5.12.8.1.1 Sequential Unformatted READ Statement

A sequential unformatted READ statement is of the form:

READ (unt [,ERR=s_1] [,END=s_2]) iolist

where: unt is a unit specifier

s_1 is a statement label for the error specifier

s_2 is a statement label for the end-of-file specifier

iolist is an input list.

5.12.8.1.1b
At the inception of execution of this statement, the sequential file connected to the specified unit must be positioned so that the next record is an unformatted record or an endfile record. It must not be positioned within a free-field record.

5.12.8.1.1c
Execution of the input statement causes reading of the next record from the sequential file connected to the specified unit. If there is a list, the values in that record are assigned to the sequence of entities specified by the list. The file is then positioned so that the record read becomes the preceding record. The number of values required by the list may be less than or equal to the number of values in the unformatted record. If the list requires more values than the record contains, an error condition exists.

5.12.8.1.1d
If the record read is an endfile record, execution of the statement causes an end-of-file condition to exist.

5.12.8.1.2 Direct Access Unformatted READ Statement

A direct access unformatted READ statement is of the form:

READ (unt, REC=rn [,ERR=s]) iolist

where: unt is an external unit specifier

rn is an integer, real, or double precision expression that has a positive value after conversion to an integer value according to the rules for arithmetic assignment statements. It specifies the number of the record that is to be read.

s is a statement label for the error specifier

iolist is an input list.

5.12.8.1.2b
Execution of this statement causes the reading of record rn from the direct access file that is currently connected to the specified unit. The values in that record are assigned to the sequence of entities specified by the list. The number of values required by the list may be less than or equal to the number of values in the record. If the list requires more values than the record contains, an error condition exists.

5.12.8.1.2c
An error condition exists if the record number rn is less than one, if rn exceeds the maximum record number for a file with the maximum-record-number property, or if rn exceeds the maximum record number given when the file was opened.

5.12.8.1.2d
Reading an undefined record causes all entities specified by the list to become undefined.

5.12.8.2 Unformatted Output Statements

5.12.8.2.1 Sequential Unformatted WRITE Statement

A sequential unformatted WRITE statement is of the form:

WRITE (unt [,ERR=s]) iolist

where: unt is an external unit specifier

s is a statement label for the error specifier

iolist is an output list. Note that the list is required.

5.12.8.2.1b
Execution of this statement creates and writes the unformatted next record of the sequential file connected to the specified unit. The record contains the sequence of values specified by the list. The unit is then positioned so that the record written is the preceding record and is also the last record in the file. If, at inception of execution of this statement, the file is positioned so that a current record exists, the record is terminated.

5.12.8.2.2 Direct Access Unformatted WRITE Statement

A direct access unformatted WRITE statement is of the form:

WRITE (unt, REC=rn [,ERR=s]) iolist

where: unt is an external unit specifier

rn is an integer, real, or double precision expression that has a positive value after conversion to an integer value according to the rules for arithmetic assignment statements. It specifies the number of the record that is to be written.

s is a statement label for the error specifier

iolist is an output list. Note that the list is required.

5.12.8.2.2b
Execution of this statement causes the writing of the sequence of values specified by the list into record rn of the direct access file that is currently connected to the specified unit. The record written becomes or remains defined.

5.12.8.2.2c
An error condition exists if the file has the formatted-record property. An error condition also exists if rn is less than one, if the file has the maximum-record-number property and rn exceeds the maximum record number for the file, or if rn exceeds the maximum record number given when the file was opened. In these cases, the file is not modified.

5.12.8.2.2d
If the list specifies more values than can fit into a record, an error condition exists. Unless another error condition exists, the record is written with as many values as will fit into it.

5.12.8.2.2e
If the values specified by the list do not fill the record, integer zero values are added to fill the record.

5.12.9 Formatted-Input/Output Statements

5.12.9.1 Formatted Input Statements

5.12.9.1.1 Sequential Formatted READ Statement

A sequential formatted READ statement is one of the forms:

READ (unt, fmt [,ERR=s_1] [,END=s_2]) iolist

READ fmt [,iolist]

where: unt is a unit specifier

fmt is a format specifier

s_1 is a statement label for the error specifier

s_2 is a statement label for the end-of-file specifier

iolist is an input list

5.12.9.1.1b
The second form, which does not have a unit specifier, is equivalent to the first form for some input unit which is preconnected by the processor.

5.12.9.1.1c
At inception of execution of this statement, the sequential file connected to the specified unit must be positioned so that the next record is a formatted record or an endfile record. It must not be positioned within a free-field record.

5.12.9.1.1d
Execution of this statement causes the reading of the next record, and possibly additional records, from the file connected to the specified unit. Each record read must be a formatted record. The information in each record is scanned and converted according to the specified format specification. The resulting values are assigned to the entities specified by the list. The file is then positioned so that the last record read becomes the preceding record.

5.12.9.1.1e
If the list and format specification require more characters than a record contains, all of the entities specified by the list become undefined and an error condition exists.

5.12.9.1.2 Direct Access Formatted READ Statement

A direct access formatted READ statement is of the form:

READ (unt, fmt, REC=rn [,ERR=s]) [iolist]

where:

unt is an external unit specifier

fmt is a format specifier

rn is an integer, real, or double precision expression that has a positive value after conversion to an integer value according to the rules for arithmetic assignment statements. It is the number of the first record that is to be read.

s is a statement label for the error specifier

iolist is an input list.

12.9.1.2b
Execution of this statement causes the reading of record rn from the direct access file that is currently connected to the specified unit. If the list and format specification require more than one record, the record number is increased by one for each additional record required and these records are read as required. The information in each record is scanned and converted according to the specified format specification. The resulting values are assigned to the entities specified by the list.

5.12.9.1.2c
If the initial record number rn is less than one, an error condition exists. If any record number exceeds the maximum record number of a file with the maximum-record-number property, or the maximum record number given when the file was opened, an error condition exists.

5.12.9.1.2d
If the list and format specification require more characters than a record contains, all of the entities specified by the list become undefined and an error condition exists.

5.12.9.1.2e
Reading an undefined record causes all entities specified by the input list to become undefined.

5.12.9.2 Formatted Output Statements

5.12.9.2.1 Sequential Formatted Output Statement

A sequential formatted output statement is of one of the forms:

WRITE (unt, fmt [,ERR=s]) [iolist]
PRINT (unt, fmt [,ERR=s]) [iolist]

WRITE fmt [,iolist]
PRINT fmt [,iolist]

where: unt is a unit specifier

fmt is a format specifier

s is a statement label for the error specifier

iolist is an output list

5.12.9.2.1b
The first form of PRINT is equivalent to the first form of WRITE. The second form of WRITE, which does not have a unit specifier, is equivalent to the first form of WRITE for some output unit which is preconnected by the processor. The second form of PRINT, which does not have a unit specifier, is equivalent to the first form of WRITE for some output unit which is preconnected by the processor. The units preconnected by the processor for WRITE and PRINT statements may be the same or different units.

5.12.9.2.1c
Execution of this statement creates and writes the formatted next record and possibly additional records on the sequential file connected to the specified unit. The list specifies a sequence of values. These are converted and positioned according to the specified format specification. The file is then positioned so that the last record written is the preceding record and is also the last record on the file. If, at inception of execution of this statement, the file is positioned so that a current record exists, that record is terminated.

5.12.9.2.2 Direct Access Formatted WRITE Statement

A direct access formatted WRITE statement is of the form:

WRITE (unt, fmt, REC=rn [,ERR=s]) [iolist]

where: unt is an external unit specifier

fmt is a format specifier

rn is an integer, real, or double precision expression that has a positive value after conversion to an integer value according to the rules for arithmetic assignment statements. It is the number of the first record that is to be written.

s_1 is a statement label for the error specifier

iolist is an output list.

5.12.9.2.2b
Execution of this statement writes the formatted record identified by rn in the direct access file that is currently connected to the specified unit. If the list and format specification specify additional records, the record number is increased by one for each additional record specified and those records are also written. All records written become or remain defined. The list specifies a sequence of values. These are converted to characters and positioned according to the specified format specification.

5.12.9.2.2c
An error condition exists if rn is less than one. If any record number exceeds the maximum record number of a file with the maximum-record-number property or exceeds the maximum record number given when the file was opened, an error condition exists. An error condition exists if this statement is executed on a unit opened with the READ ONLY option, in which case the contents of the file is not modified.

5.12.9.2.2d
If the values specified by the list and format do not fill a record, blank characters are added to fill the record.

5.12.9.2.2e
If the list and format specification specify more characters than can fit into a record, an error condition exists. Unless another error condition exists, the record is written with as many characters as will fit.

5.12.9.3 Printing of Formatted Records

If a formatted record is printed, the first character of the record is not printed. The remaining characters of the record, if any, are printed in one line beginning at the left margin.

5.12.9.3b
The first character of such a record determines vertical spacing as follows:

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

5.12.9.3c
If there are no characters in the record, the vertical spacing is one line and no characters other than blank are printed in that line.

5.12.9.3d
WRITE and PRINT do not imply that printing either will or will not take place.

5.12.10 List-Directed Input/Output Statements

List-directed input/output statements are used to read and write free-field records of sequential files.

5.12.10.1 Free-field Records

A free-field record consists of a sequence of values composed of characters which are capable of representation in the processor. The forms of the values are described in 5.12.10.2.1 and 5.12.10.3.1. Values within a free-field record are read and written only by list-directed input/output statements. The length of a free-field record is measured in values and is not necessarily known to the processor. Note that it is not possible to have two consecutive free-field records. In addition, free-field records have the following properties:

(1) The values comprising the record exist as a totally ordered set. There is a first value and a last value. The order of the values is the order in which they were written.

(2) The record has a position property, from which are derived the related concepts of next value, preceding value, and initial point.

(3) If a record is positioned between the ith and ith+1 values, then the ith value is the preceding value and the ith+1 value is the next value.

(4) The initial point of a record is that unique position where the next value is the first value in the record. If a file is positioned so that the next record is a free-field record, that record is positioned at its initial point. When the record is positioned at its initial point, there is no preceding value.

(5) When the record is positioned after the last value, the last value is the preceding value and there is no next value.

5.12.10.1b
A free-field record consists of all the characters comprising a sequence of values which can be read by one or more successive list-directed input statements or which have been written by one or more successive list-directed output statements. However, it is recognized that essentially all processors will subdivide a long sequence of characters into shorter sequences of characters that appear as separate lines when printed. Therefore, a free-field record also consists of one or more lines. The length of a line is measured in characters and may vary among processors and among devices on the same processor.

5.12.10.2 List-Directed READ Statement

A list-directed READ statement is of one of the forms:

READ (unt, star [,ERR=s_1] [,END=s_2]) iolist

READ star, iolist

where: unt is an external unit specifier

star is either the character * or the form FMT=*

s_1 is a statement label for the error specifier

s_2 is a statement label for the end-of-file specifier

iolist is an input list. Note that the list is required.

5.12.10.2b
The second form, which does not have a unit specifier, is equivalent to the first form for some input unit which is preconnected by the processor.

5.12.10.2c
Upon inception of execution of a list-directed READ statement, the file connected to the specified unit must be positioned so that the next or current record is a free-field record containing values conforming to 5.12.10.2.1, or so that the next record is an endfile record.

5.12.10.2d
If the next or current record is a free-field record, execution of the READ statement causes one or more successive values, beginning with the next value, to be read from the record and assigned to the sequence of entities specified by the list. The record is then positioned so that the last value read is the preceding value. If this new preceding value is not the last value of the record, the record becomes or remains the current record. If the new preceding value is the last value of the record, the record becomes the preceding record and there is no current record.

5.12.10.2e
If an endfile record is read during execution of a list-directed READ statement, an end-of-file condition exists.

5.12.10.2f
Note that each READ statement does not always begin reading data at the beginning of a line and that reading may continue with the remaining values of a repeated constant that supplied some values to a previous READ statement.

5.12.10.2.1 Free-Field Input Records

A free-field input record consists of a sequence of values and value separators. Values are separated by one of the following four kinds of value separators:

(1) one or more blanks

(2) a comma optionally preceded by one or more blanks and optionally followed by one or more blanks

(3) a slash optionally preceded by one or more blanks and optionally followed by one or more blanks

(4) the end of a line, except within a character constant or complex constant.

5.12.10.2.1b
Each value is either a constant, a null value, or one of the forms:

r*c
r*

where r is an unsigned nonzero, integer constant. The r*c form is equivalent to r successive appearances of the constant c and the r* form is equivalent to rsuccessive null values. Neither of these two forms may contain embedded blanks, except where permitted within the constant c. Neither may contain an imbedded end of a line.

5.12.10.2.1c
Hollerith constants are not permitted in free-field input records. Otherwise any form acceptable in a formatted input field for a given type (Section 5.13) is also acceptable as a free-field input value. In addition, an integer constant is acceptable as a real constant with an implied decimal point to the right of the rightmost digit. The form of the input value must be acceptable for the type of the list item. Blanks are never used as zeros, and embedded blanks are not permitted in constants except within character constants and surrounding the comma between the real and imaginary parts of a complex constant.

5.12.10.2.1d
A null value, as specified by the r* form, the first nonblank character being a slash or a comma, two commas separated by zero, one, or more blanks, or a comma and a slash in either order separated by zero, one, or more blanks, has no effect on the definition or undefinition of the corresponding list item. If the list item is defined, it retains its previous value, if it is undefined, it remains undefined. A null value may not be used as either the real or imaginary part of a complex constant, but it may be supplied in place of an entire complex constant. Note that the end of a line adjacent to any other separator, with or without separating blanks, does not generate a null item.

5.12.10.2.1e
A slash encountered in the input record causes termination of execution of the statement after the assignment of any previous values. If there are additional items in the input list. the effect is as though null values had been supplied for them.

5.12.10.2.1f
Note that a complex constant must have its normal form of a pair of real or integer constants enclosed in parentheses. Embedded blanks are permitted immediately before or after the comma that separates the real part and the imaginary part, and the end of a line may occur between the real part and the comma or between the comma and the imaginary part.

5.12.10.2.1g
Hollerith constants are not permitted in free-field input records. Character constants enclosed in apostrophes are permitted when the corresponding list item is of type character. Each apostrophe within a character constant must be represented by two consecutive apostrophes with neither blanks nor the end of a line intervening. Character constants may be continued from the end of one line to the beginning of the next line. The end of the line does not cause a blank or any other character to become part of the constant. The constant may be continued on as many lines as needed. Blanks, commas, and slashes may appear in character constants.

5.12.10.3 List-Directed Output Statement

A list-directed output statement is of one of the forms:

```
WRITE (unt, star [, ERR=s]) iolist
PRINT (unt, star [, ERR=s]) iolist

WRITE star, iolist
PRINT star, iolist
```

where: unt is an external unit specifier

star is either the character * or the form FMT=*

s is a statement label for the error specifier

iolist is an output list. Note that the list is required.

5.12.10.3b
The second form of WRITE, which does not have a unit specifier, is equivalent to the first form of WRITE for some output unit which is preconnected by the processor. The second form of PRINT, which does not have a unit specifier, is equivalent to the first form of PRINT for some output unit which is preconnected by the processor. The units preconnected by the processor for WRITE and PRINT statements may be the same or different units.

5.12.10.3c
Execution of a list-directed output statement created or extends a free field record of the sequential file connected to the specified unit.

5.12.10.3d
If, at inception of execution of the output statement, the file is positioned at its initial point, execution of the statement creates the first record. If, at inception of execution, the file is positioned so that the ith record is the preceding record and there is no current record, execution creates the ith+1 record. In either case, execution of the output statement causes the sequence of items specified by the list to be written as a sequence of values beginning with the first value of the record. The values produced must conform to 5.12.10.3.1. The record is positioned so that the last value written is the preceding value and is also the last value in the record. The record created becomes the current record and is the last record in the file.

5.12.10.3e
If, at inception of execution of the output statement, there is a current record, execution of the output statement extends that record. Execution of the output statement causes the sequence of elements specified by the list to be written on the record as a sequence of values beginning with the next value of the record. The values produced must conform to 5.12.10.3.1. The file is positioned so that the last value written is the preceding value and is also the last value in the record. The record remains the current record and is the last record in the file. Note that an output statement does not necessarily begin a new record. However, inception of execution of an output statement begins a new line.

5.12.10.3f
If, at inception of execution of any sequential or list-directed output statement, the file is positioned within a repeated constant, the repeat count of that constant is reduced by the processor in order to preserve the contents of the record read up to that point. Note that this may require modification and rewriting of the most recent input line, but no others, since the form r*c must be contained within a single line.

5.12.10.3.1 Free-field Output Records

A free-field output record consists of a sequence of values that have the same form as constants, except as noted otherwise. The only separators produced are blanks and the end of a line. The processor may start new lines as necessary, but, except for complex constants and character constants, the end of a line must not occur within a constant and blanks must not appear within a constant.

5.12.10.3.1b
Logical output constants are T for a true value and F for a false value.

5.12.10.3.1c
Real and double precision constants are produced with either the equivalent of an F field descriptor or an E field descriptor, depending on the value and the processor. For a value of x and some processor-dependent integer values for d_1 and d_2, the constant is produced with some reasonable F field descriptor if $10^{**}d_1 \leq |x| < 10^{**}d_2$; otherwise, a field descriptor of 1PEw.dEe is used for some reasonable values of w, d, and e, except that the scale factor does not affect any subsequent fields.

5.12.10.3.1d
Complex constants are enclosed in parentheses, with a comma separating the real and imaginary parts. The end of a line may occur between the comma and the imaginary part only if the entire constant is longer than an entire line. The only embedded blanks permitted within a complex constant are between the comma and the end of a line and one blank at the beginning of the next line.

5.12.10.3.1e
Hollerith constants are not produced in free-field output records. The end of a line may not occur within a character constant unless the constant is longer than an entire line. Note that such constants cannot be repeated by use of the form r*c. Character constants produced by a PRINT statement are not delimited by apostrophes, are not preceded or followed by a value separator, have each internal apostrophe represented externally by one apostrophe, and have a blank character inserted by the processor for carriage control at the beginning of any line that begins with the continuation of a character constant from the previous line. Character constants produced by WRITE statements are delimited externally by apostrophes, are preceded and followed by a value separator, have each internal apostrophe represented externally by two successive apostrophes without blanks or the end of a line intervening, and do not have a carriage control character inserted by the processor when a character constant is continued from one line to the next. Note that free-field records produced by PRINT statements are not always acceptable as free-field input records, but free-field records produced by WRITE statements are always acceptable as free-field input records.

5.12.10.3.1f
If two or more successive values in a free-field output record produced by a list-directed PRINT statement have identical values, the processor has the option of producing a repeated constant of the form r*c instead of the sequence of identical values. The processor does not have this option for records produced by list-directed WRITE statements.

5.12.10.3.1g
Commas and slashes as separators, and null values (including the form r*), are not produced in free-field output records.

5.12.10.3.1h
Except for character constants continued from the previous line by a WRITE statement, each free-field output line begins with a blank character to provide carriage control when the line is printed.

5.12.11 Auxiliary Input/Output Statements

5.12.11.1 OPEN Statement

An OPEN statement may be used to connect an existing file to a unit, or to create and connect a file to a unit.

5.12.11.1b
An OPEN statement is of the form:

```
OPEN(unt [,NAME=fin] [,STATUS=typ] [,ACCESS=acc]
     [,FORM =ft] [,RECL=rl] [,MAXREC=maxr]
     [,ERR=s] [,BLANK=blnk] [,READ ONLY] )
```

where: unt is an external unit specifier

fin is a character expression that is the name of the file to be connected to the specified unit. An error condition exists if the file name is not acceptable and meaningful to the processor. If this specifier is omitted, the specified unit is connected to a processor-determined file. Note that this processor-determined file must not be preconnected.

typ is a character expression whose value when any trailing blanks are removed is 'OLD', 'NEW', 'SCRATCH', or 'UNKNOWN'. If 'OLD' or 'NEW' is specified, a NAME specifier must be given. If 'OLD' is specified, the file must exist; otherwise, an error condition exists. If 'NEW' is specified, the file must not exist; otherwise, an error condition exists. If 'SCRATCH' is specified with an unnamed file, the file is connected to the specified unit for use by the executable program but is deleted either at the execution of a CLOSE statement referring to the same unit or at the termination of the executable program. 'SCRATCH' may not be specified with a named file. If 'UNKNOWN' is specified, then the type is processor dependent. If typ is not specified, a value of 'UNKNOWN' is assumed.

5.12.11.1b2

acc is a character expression whose value when any trailing blanks are removed is 'DIRECT' or 'SEQUENTIAL'. It specifies the file as being a direct access file or a sequential file. For an existing file, this specification must agree with the actual file; otherwise, an error condition exists. If this specifier is omitted, the assumed value is the same as the access method for an existing file, and is 'SEQUENTIAL' for files being created.

ft is a character expression whose value when any trailing blanks are removed is 'FORMATTED' or 'UNFORMATTED'. It may be specified only for a direct access file. If the file is being created, this specifier determines whether the records of the file are all formatted, or all unformatted. If the file exists, the specifier must agree with the actual property of the file; otherwise, an error condition exists. If this specifier is omitted for an existing file, the actual property of the file will be assumed. If this specifier is omitted for a file being created, the value 'UNFORMATTED' is assumed.

rl is an integer, real, or double precision expression, whose value is converted to an integer according to the rules for arithmetic assignment statements. It specifies the length of each record in a direct access file. If the records are formatted, the length is the number of characters. If the records are unformatted, the length is the number of storage units. If the file exists and rl does not agree with the actual length of the records, an error condition exists. If the file is being created, rl specifies the record length property. If this specifier is omitted for old files, the actual record length is assumed. This specifier must be given when a direct access file is created. This specifier may not be given for a sequential file.

5.12.11.1b3

maxr is an integer, real, or double precision expression, whose value is converted to an integer according to the rules for arithmetic assignment statements. If MAXREC=maxr is specified when a file is created, the file is given the maximum-record-number property, and maxr is the maximum record number. If this specifier is omitted when the file is created, the file is given the property of having no maximum record number, and the MAXREC=maxr specifier may not be given on subsequent openings of the file. An error condition exists if maxr is specified for a file that is not a direct access file, or for an existing direct access file that does not have the maximum-record-number property. For existing files with the maximum-record-number property, an error condition exists if maxr is specified to be other than the maximum record number of the file. It may be specified to be less than the maximum record number of the file. In this case, maxr does not modify the property of the file, but does become the maximum record number of the file that can be read or written by this program. An attempt to read or write using a larger record number causes an error condition to exist and results in no modification to the file. If this specifier is omitted on an existing file with the maximum-record-number property, the assumed value is the maximum record number of the file.

s is a statement label for the error specifier

blnk is a character expression whose value when any blanks are removed is 'BLANK' or 'ZERO'. If 'BLANK' is specified all blank characters in numeric formatted input fields on the specified unit are ignored, except that a field of all blanks has a value of zero. If 'ZERO' is specified, all blanks other than leading blanks are treated as zeros. If this specifier is omitted, then the effect is as if BLANK='ZERO' were specified.

5.12.11.1c
The READ ONLY specifier permits reading of an old file and prohibits writing into that file. Execution of an output statement on the unit results in error condition and results in no modification to the file. Note that this specifier does not assign an property to the file, but only protects it against accidental modification by the executable program.

5.12.11.1d
The unit specifier is required to appear; all other specifiers are optional, except that the record length rl must be specified if the file is being created and the access is direct. As noted above, however, some of the specifications have an assumed value if they are omitted.

5.12.11.1e
If a unit is connected to a file, the execution of an OPEN statement on that unit, or on that file, is not permitted. The previous connection could be by a previous execution of an OPEN statement or by preconnection.

5.12.11.1f
A unit may be connected by execution of an OPEN statement in any program unit of an executable program and, once connected, may be referenced in any program unit of that executable program.

5.12.11.2 CLOSE Statement

A CLOSE statement may be used to remove the connection of a particular file or a sequence of sequential files to a unit.

5.12.11.2b
A CLOSE statement is of the form:

CLOSE (unt [,STATUS=dis] [,ERR=s])

where: unt is an external unit specifier

dis is a character expression whose value when any trailing blanks are removed is 'KEEP' or 'DELETE'. dis determines the disposition of the file that is connected to the specified unit.

If 'KEEP' is specified, the file continues to exist after the execution of the CLOSE statement. If 'DELETE' is specified, the file ceases to exist after execution of the CLOSE statement, unless it was opened with the READ ONLY option. If the file was opened with the READ ONLY option and 'DELETE' is specified, an error condition exists and the file continues to exist after execution of the CLOSE statement. If this specifier is omitted, the assumed value is 'KEEP' unless the file's type is 'SCRATCH' in which case the assumed value is 'DELETE'.

s is a statement label for the error specifier.

5.12.11.2c
The closing of a unit may occur in any program unit of an executable program and need not be done in the same program unit as the opening.

5.12.11.2d
Execution of a CLOSE statement referring to a unit that has no file connected to it is permitted and has no effect.

5.12.11.2e
After a unit has been disconnected by execution of a CLOSE statement, it may be connected again within the same executable program and connected to the same file or a different file. After a file has been disconnected by execution of a CLOSE statement, it may be connected again within the same executable program to the same or to a different unit.

5.12.11.3 INQUIRE Statement

An INQUIRE statement may be used to inquire about properties of a particular named file or of the file connected to a particular unit. There are two forms of the INQUIRE statement: inquire by file and inquire by unit. Note that, except for the specifiers that identify the two forms, they contain the same set of specifiers which identify variables or array elements in which values for the properties are to be returned. However, the values of some of these properties have different meanings, depending upon which form is used.

5.12.11.3.1 INQUIRE by File

The INQUIRE by file statement is of the form:

```
INQUIRE ( FILE=fin    [,EXIST=ex]    [,OPENED=op]
        [,NUMBER=num] [,NAMED=nmd]   [,NAME=fn]
        [,MAXREC=maxr] [,ERR=s] )
```

where: fin is a character expression specifying the name of the file being inquired about. The named file may or may not be connected to a unit. If the value of fin is not of a form acceptable to the processor as a file name, an error condition exists.

ex is a logical variable or logical array element that becomes defined to true if there exists a file by that name; otherwise, ex becomes defined to false.

op is a logical variable or logical array element that becomes defined to true if the file is connected to a unit; otherwise, op becomes defined to false.

5.12.11.3.1b
The specifier variables or array elements num, nmd, fn, acc, fm, rcl, and maxr, become defined only if the value of fin is acceptable to the processor as a file name and if there exists a file by that name; otherwise, they become undefined. They are described in 5.12.11.3.3. s is also described in 5.12.11.3.3.

5.12.11.3.2 INQUIRE by Unit

The INQUIRE by unit statement is of the form:

```
INQUIRE ( UNIT=u    [,EXIST=ex]    [,OPENED=op]
        [,NUMBER=num]  [,NAMED=nmd]  [,NAME=fn]
        [,MAXREC=maxr] [,ERR=s] )
```

where: u is an external unit identifier. It identifies a unit to which a file need not be connected. If a file is connected to this unit, it is the file being inquired about.

ex is a logical variable or logical array element that becomes defined to true if the specified unit exists and is known to the processor; otherwise, ex becomes defined to false.

op is a logical variable or logical array element that becomes defined to true if a file is connected to the unit; otherwise, op becomes defined to false.

5.12.11.3.2b
The specifier variables or array elements num, nmd, fn, acc, fm, rcl, and maxr become defined only if the specified unit exists and is known to the processor and if a file is connected to that unit; otherwise, they become undefined. They are described in 5.12.11.3.3. s is also described in 5.12.11.3.3.

5.12.11.3.3 Inquiry Specifiers for Either Form

Either form of the INQUIRE statement may include inquiry specifiers containing the following entities:

nmd — is a logical variable or logical array element that becomes defined to true if the file has a name; otherwise, it becomes defined to false.

fn — is a character variable or character array element that becomes defined to the name of the file, if the file has a name; otherwise, fn becomes undefined. Note that, if this specifier appears in an inquire by file, its value is not necessarily the same as the name given in the FILE=fin specifier.

num — is an integer variable or integer array element that becomes defined to the value of the external unit identifier of the unit that is currently connected to the file. If there is no unit connected to the file, num becomes undefined.

acc — is a character variable or character array element that becomes defined to 'DIRECT' if the file is a direct access file and becomes defined to 'SEQUENTIAL' if the file is a sequential access file.

fm — is a character variable or character array element that becomes defined to 'FORMATTED' if the direct access file consists entirely of 'FORMATTED' records and becomes defined to 'UNFORMATTED' if that file consists entirely of unformatted records. If the file is not a direct access file, fm becomes undefined.

rcl — is an integer variable or integer array element that becomes defined to the record length of the direct access file. The length is measured in characters for formatted files and in storage units for unformatted files. rcl becomes undefined if the file is not a direct access file.

maxr — is an integer variable or integer array element that becomes defined to the maximum number of records in the direct access file. If the file does not have the maximum-record-number property, maxr is undefined.

s — is a statement label for the error specifier. If an error condition occurs during execution of an INQUIRE statement, all the specified variables and array elements become undefined.

5.12.11.3.3b
An INQUIRE statement may be executed before, while, or after a file is connected to a unit. All definitions caused by the INQUIRE statement are those that are current at the time the statement is executed.

5.12.11.4 REWIND Statement

A REWIND statement is of one of the forms:

REWIND u

REWIND (unt [,ERR=s])

where: u is an external unit identifier

unt is an external unit specifier

s is a statement label for the error specifier.

5.12.11.4b
Execution of this statement causes the file connected to the specified unit to be positioned at its initial point. If the file connected to the specified unit is already positioned at its initial point, execution of this statement has no effect.

5.12.11.4c
The specified unit must be connected to a sequential file.

5.12.11.5 BACKSPACE Statement

A BACKSPACE statement is of one of the forms:

BACKSPACE u

BACKSPACE (unt [,COUNT=n] [ERR=s])

where: u is an external unit identifier

unt is an external unit specifier

n is an integer, real, or double precision expression that has a positive or zero value after conversion to an integer value according to the rules for arithmetic assignment statements.

s is a statement label for the error specifier.

5.12.11.5b
The file connected to the specified unit is repositioned as follows: BACKSPACE without COUNT=n causes the preceding record, if it exists, to become the next record. If there is no preceding record, execution of the statement has no effect. BACKSPACE with COUNT=n is equivalent to n consecutive applications of BACKSPACE without COUNT=n. Note that endfile records are counted as records during execution of this statement. Note also that n=0 results in no repositioning.

5.12.11.5c
Backspacing over free-field records is not permitted.

5.12.11.5d
The specified unit must be connected to a sequential file.

5.12.11.6 ENDFILE Statement

An ENDFILE statement is of one of the forms:

ENDFILE u

ENDFILE (unt [,ERR=s])

where: u is an external unit identifier

unt is an external unit specifier

s is a statement label for the error specifier

Execution of this statement creates an endfile record as the next record of the file. The unit is then repositioned so that the record written is the preceding record, and is also the last record in the file. If at inception of execution of this statement, the file is positioned so that a current record exists, that record is terminated by inception of execution of the statement.

5.12.11.6b
The specified unit must be connected to a sequential file.

5.12.11.7 BACKFILE Statement

A BACKFILE statement is of one of the forms:

BACKFILE u

BACKFILE (unt [,COUNT=n] [,ERR=s])

where: u is an external unit identifier

unt is an external unit specifier

n is an integer, real, or double precision expression that has a positive or zero value after conversion to an integer value according to the rules for arithmetic assignment statements.

s is a statement label for the error specifier.

5.12.11.7b
The sequence of sequential files connected to the specified unit is repositioned as follows: BACKFILE without COUNT=n causes the endfile record of the preceding

file, if it exists, to become the next record. If there is no preceding file, the unit is positioned at its initial point. BACKFILE with COUNT=n is equivalent to n applications of BACKFILE without COUNT=n. If n=0, no repositioning occurs.

5.12.11.7c
The specified unit must be connected to a sequential file.

5.12.11.8 SKIPFILE Statement

A SKIPFILE statement is of one of the forms:

SKIPFILE u

SKIPFILE (unt [,COUNT=n] [,ERR=s])

where: u is an external unit identifier

unt is an external unit specifier

n is an integer, real, or double precision expression that has a positive or zero value after conversion to an integer value according to the rules for arithmetic assignment statements.

s is a statement label for the error specifier.

5.12.11.8b
SKIPFILE without COUNT=n causes the sequence of sequential files connected to the specified unit to be repositioned so that the next endfile record becomes the preceding record. This form cannot be used unless an endfile record exists beyond the current position of the file. SKIPFILE with COUNT-n is equivalent to n applications of SKIPFILE without COUNT=n. This form may not be used unless at least n endfile records exist beyond the current position of the file. If n=0, no repositioning occurs.

5.12.11.8c
The specified unit must be connected to a sequential file.

5.12.12 Restrictions on Function References and List Items

A function cannot be referenced within an expression appearing anywhere in an input/output statement if such a reference causes any input/output statement to be executed. A list item, or a function reference appearing in a list item, may not affect any unit identification, format specification, or record specifier that appears in the same input/output statement. Similarly, a function reference that appears in a unit specifier, format specifier, or record specifier may not affect any entity appearing in the input/output statement except the entity that contains the function reference. A function reference that appears in a list item may affect subsequent list items, but not prior list items.

file, if it exists, to become the next record. If there is no preceding file, the unit is positioned at its initial point. BACKFILE with COUNT=n is equivalent to n applications of BACKFILE without COUNT=n. If n=0, no repositioning occurs.

5.12.11.7c
The specified unit must be connected to a sequential file.

5.12.11.8 SKIPFILE Statement

A SKIPFILE statement is of one of the forms:

SKIPFILE u

SKIPFILE (unt [,COUNT=n] [,ERR=s])

where: u is an external unit identifier

unt is an external unit specifier

n is an integer, real, or double precision expression that has a positive or zero value after conversion to an integer value according to the rules for arithmetic assignment statements.

s is a statement label for the error specifier.

5.12.11.8b
SKIPFILE without COUNT=n causes the sequence of sequential files connected to the specified unit to be repositioned so that the next endfile record becomes the preceding record. This form cannot be used unless an endfile record exists beyond the current position of the file. SKIPFILE with COUNT-n is equivalent to n applications of SKIPFILE without COUNT=n. This form may not be used unless at least n endfile records exist beyond the current position of the file. If n=0, no repositioning occurs.

5.12.11.8c
The specified unit must be connected to a sequential file.

5.12.12 Restrictions on Function References and List Items

A function cannot be referenced within an expression appearing anywhere in an input/output statement if such a reference causes any input/output statement to be executed. A list item, or a function reference appearing in a list item, may not affect any unit identification, format specification, or record specifier that appears in the same input/output statement. Similarly, a function reference that appears in a unit specifier, format specifier, or record specifier may not affect any entity appearing in the input/output statement except the entity that contains the function reference. A function reference that appears in a list item may affect subsequent list items, but not prior list items.

5.12.13 Restriction on Input/Output Statements

If a unit, or a file connected to a unit, does not have all of the properties required for the execution of certain input/output statements, those statements may not refer to that unit. For example, an input statement may not specify a unit that cannot provide input to the processor.

5.12.14 ENCODE/DECODE STATEMENTS

The ENCODE and DECODE statements provide the capability of making memory-to-memory data transfers under format control. The object code generated by the compiler for these statements is similar to that generated for the formatted WRITE and READ statements.

The general forms of the statements are

ENCODE (block, f) list

DECODE (block, f) list

In the above statements, list represents a standard I/O list, f is the statement number of a FORMAT statement or the name of an array that contains format specifications, and block is the name of an array or variable to/from which data is to be transferred.

Execution of the ENCODE statement causes the contents of list to be converted according to the specified format and the results to be stored in block. The string of characters generated are stored, as 7-bit ASCII, into consecutive locations of block. A block into which data is ENCODE'd must be at least 132 bytes in length.

Execution of the DECODE statement causes the contents of the block to be moved into the list items in accordance with the format specifications.

The number of characters per logical record is limited to 132 characters. If the number of characters in a logical record generated by ENCODE is not a multiple of four, the last word is blank filled. For both ENCODE and DECODE, the image of a subsequent record starts with the first character of the next word in the block area.

The ENCODE and DECODE statements can be likened to formatted WRITE and READ statements. Rather than transferring data between a peripheral unit and main storage, data is transferred between the areas of main storage. Thus, it is possible to move information from block to list while manipulating it with format specifications without accessing a peripheral device.

5.12.15 INCLUDE Statement

The INCLUDE statement is a compile time statement and is of the form:

INCLUDE (unt, filename)

This statement will cause the compiler to read its source from the filename on logical unit unt. When the compiler reaches an END statement or an end-of-file on the file, the compiler resumes reading from the original input source. The file on unit unt may not contain an include statement.

5.13 FORMAT SPECIFICATION

Format specifications are used in conjunction with formatted input/output statements to provide conversion and editing information between the internal representation and the external character strings. Format specifications may be given:

(1) using FORMAT statements.

(2) as values of character variables, character arrays, or character expressions, and

(3) as Hollerith data in arrays of type other than character.

5.13.1.1 Character Format Specifiations

When the format identifier (12.4) in any of the formatted input/output statements is a character expression, the first part of the specified entity must contain character data that constitutes a format specification.

5.13.1.1b
The format specification must have the form described in 13.2. It must begin with a left parenthesis and end with a right parenthesis. Character data following the right parenthesis that ends the format specification has no effect on the format specification.

A character format specification must not contain a Hollerith field descriptor; it may contain a character field descriptor.

5.13.1.1c
If the format identifier is a character array, the length of the format specification may exceed the length of the first element of that array; a character array format identifier is considered to be a concatenation of all of the array elements of that array. However, if a character array element is specified as a format identifier, the length of the format specification must not exceed the length of that array element.

5.13.1.2 FORMAT Statements

A FORMAT statement is of the form:

FORMAT fs

where: fs is a format specification as described in 5.13.2. The statement must be labeled.

5.13.1.3 Hollerith Format Specifications

When the format identifier in a formatted input/output statement is an array name of type other than character, the first part of the

specified entity must contain Hollerith data that constitutes a format specification.

5.13.1.3b
The format specification must be of the form described in 5.13.2. It must begin with a left parenthesis, and end with a right parenthesis. Hollerith data following the right parenthesis that ends the format specification has no effect on the format specification.

5.13.1.3c
A Hollerith format specification cannot contain an apostrophe field descriptor or a Hollerith field descriptor.

5.13.1.3d
Hollerith format specifications may be inserted in arrays only by the use of DATA statements, or by the use of READ statements with Aw field descriptors.

5.13.2 Format Specification

A format specification is a format group or is the form (). A format group is of the form:

(a, t, z, t, z ... z, t, a)

where: a is a series of one or more slashes, a colon, a series of slashes with a single colon preceding, following, or embedded, or is empty.

t is a field descriptor or an optionally repeated format group.

z is a field separator.

An optionally repeated format group is of the form:

[r] format group

where r is a nonzero, unsigned, integer constant. If r is omitted, it assumed value is 1.

5.13.2.1 Field Descriptors

Field descriptors are of the forms:

[s][r] Fw.d
[s][r] Ew.d
[s][r] Ew.dEe
[s][r] Ew.dDe
[s][r] Gw.d
[s][r] Dw.d
[r] Iw
[r] Iw.m
[r] Lw

[r] Aw
nH $h_1 h_2 \ldots h_n$
'$h_1 h_2 \ldots h_n$'
r X
Tc
kP
[±] S

where: F, E, G, D, I, L, A, H, X, T, P, and S are letters that indicate the manner of conversion and editing between the internal and external representations and are called the conversion codes.

W, n, e, c, and r are nonzero, unsigned, integer constants.

k is an optionally signed integer constant.

d and m are unsigned integer constants.

s represents a scale factor designator, described in 5.13.3.1.2

h is one of the characters capable of representation by the processor.

n is a nonzero, optionally signed, integer constant.

± signifies a plus or a minus

5.13.2.1b
Constants in format specifications may not be symbolic names of constants (8.7).

5.13.2.1c
The phrase basic field descriptor is used to signify the field descriptor unmodified by s or r.

5.13.2.1d
The internal representation of external fields corresponds to the internal representation of the corresponding type constants (Section 4).

5.13.2.2 Field Separators

A format field separator is a comma, a colon, a series of one or more slashes, or a series of slashes with a single colon preceding, following or embedded. The field descriptors or groups (13.2.3) of field descriptors are separated by a field separator.

5.13.2.2b
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 length of a formatted record depends primarily upon the number of characters put into the record when it was written. However, it may be dependent upon the processor and the external medium.

5.13.2.2c
The processing of the number of characters that can be contained in a record by an external medium does not of itself cause the beginning of processing of the next record.

5.13.2.2d
The colon is used not only to separate field descriptors, but to terminate format control (13.4) if there are no more items in the input/output list.

5.13.2.3 Repeat Specifications

Repetition of the field descriptors (except H, X, T, S, P, and apostrophe field descriptors) or format group is accomplished by using the repeat count. If the input/output list is long enough, the specified basic field descriptor or format group will be interpreted repetitively the specified number of times.

5.13.3 Basic Field Descriptors

5.13.3.1 Scale Factor

A scale factor may be specified by a separate P field descriptor or by a scale factor designator.

5.13.3.1.1 P Field Descriptor

The P field descriptor is of the form:

kP

where k, the scale factor, is an optionally signed integer constant.

5.13.3.1.2 Scale Factor Designator

A scale factor designator may be used with the F, E, G, and D conversions and is of the form:

kP

where k, the scale factor, is an optionally signed integer constant.

5.13.3.1.3 Scale Factor Effects

At beginning of execution of a formatted input/output statement, a scale factor of zero is established. Once a scale factor has been established, it applies to all subsequently interpreted F, E, G, and D field descriptors until another scale factor is encountered, and then that scale factor is established.

5.13.3.1.3b
The scale factor k affects the appropriate conversions in the following manner:

(1) For F, E, G, and D 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 multiplied by 10 ** k.

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

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

(4) For G output, the effect of the scale factor is suspended unless the magnitude of the datum to be converted is outside the range that permits the use of F conversion. If the use of E conversion is required, the scale factor has the same effect as with E output.

5.13.3.2 Numeric Conversions

The numeric field descriptors I, F, E, G, and D are used to specify input/output of integer, real, double precision, and complex data. Except where noted otherwise, the following general rules apply:

(1) With all numeric input conversions, leading blanks are not significant and other blanks are zero unless an executed OPEN statement has specified that all blanks are insignificant for the file being read. Plus signs may be omitted. A field of all blanks is considered to be zero.

(2) With the F, E, G, and D input conversions, a decimal point appearing in the input field overrides the decimal point specification supplied by the field descriptor. The input field may have more digits than the processor will use to approximate the value of the datum.

(3) With all output conversions, the external representation of a negative value must be signed. The external representation of a positive or zero value may have a sign, or may not, as controlled by the S field descriptor (13.3.8) or the processor.

(4) 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.

(5) If the number of characters produced by an output conversion exceeds the field width, or if an output exponent exceeds its specified length, the processor will fill the entire field of width w with asterisks. However, the processor must not produce asterisks if the field width is not exceeded when optional characters are omitted.

5.13.3.2.1 Integer Conversions

The numeric field descriptors Iw and Iw · m indicate that the external field occupies w positions as an integer. The value of thelist item appears, or is to appear, internally as an integer datum.

5.13.3.2.1b
In the external input field, the character string must be in the form of an optionally signed integer constant (4.2.1), except for the interpretation of blanks (13.3.2, item (1)).

5.13.3.2.1c
The external output field for the Iw field descriptor 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 unsigned integer constant. Note that an integer constant always consists of at least one decimal digit.

5.13.3.2.1d
The external output field for the Iw·m field descriptor is the same as for the Iw field descriptor except the unsigned integer constant consists of at least m decimal digits and, if necessary, has leading zeros. If m is zero and the value of the internal datum is zero, the external output field consists of only blank characters.

5.13.3.2.2 Real and Double Precision Conversions

There are four conversions available for use with real and double precision data: F, E, G, and D.

5.13.3.2.2.1 F Conversion

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

5.13.3.2.2.1b
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 optionally signed integer constant.

(3) D followed by an optionally signed integer constant.

An exponent containing D is equivalent to an exponent containing E.

5.13.3.2.2.1c
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 a string of digits containing a decimal point representing the magnitude of the internal value, as modified by the established scale factor, and rounded to d fractional digits.

5.13.3.2.2.2 E and D Conversion

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

5.13.3.2.2.2b
The form of the external input field is the same as for the F conversion.

5.13.3.2.2.2c
The form of the external output field for a scale factor of zero is:

[±] [0 . $x_1 x_2 \ldots x_d$ exp [for n read /]

where:

± signifies a plus or a minus.

$x_1, x_2 \ldots x\_sub\_d$ are the d most significant rounded digits of the value of the data to be produced.

exp is a decimal exponent of one of the following forms:

Field Descriptor	Absolute Value of Exponent	Form of Exponent
Ew·d	≤ 99	$E\pm y_1y_2$ or $\pm y_1y_2y_3$
	$99 < \lvert exp \rvert \leq 999$	$\pm y_1y_2y_3$
Dw·d	≤ 99	$D\pm y_1y_2$ or $E\pm y_1y_2$
	$99 < \lvert exp \rvert \leq 999$	or $\pm y_1y_2y_3$ $\pm y_1y_2y_3$
Ew·dEe	$\leq 10**e - 1$	$E\pm y_1y_2 \ldots y\_sub\_e$
Ew·dDe	$\leq 10**e - 1$	$D\pm y_1y_2 \ldots y\_sub\_e$

y is a decimal digit; the sign in the exponent is required.

5.13.3.2.2.2d
The scale factor k controls the decimal normalization between the number part and the exponent part (13.3.1). If $-d < k \leq 0$, there will be exactly -k leading zeros and d+k significant digits after the decimal point. If $0 < k < d + 2$, there will be exactly k significant digits to the left of the decimal point and d-k+1 significant digits to the rigth of the decimal point.

5.13.3.2.2.3 G Conversion

The numeric field descriptor Gw·d indicates that the external field occupies w positions with d significant digits. If the list item is real, the value appears, or is to appear, internally as a real datum. If the list item is double precision, the value appears, or is to appear, internally as a double precision datum.

5.13.3.2.2.3b
Input processing is the same as for F conversion.

5.13.3.2.2.3c
The method of representation in the external output field depends on the magnitude of the real datum being converted. Let N be the magnitude of the internal datum. The value of N determines the conversion as follows:

Magnitude of Datum	Equivalent Conversion
$0.1 \leq N < 1$	F(w-4).d,4X
$1 \leq N < 10$	F(w-4).(d-1),4X
.	.
.	.
.	.
10**(d-2) $\leq$ N < 10**(d-1)	F(w-4).1,4X
10**(d-1) $\leq$ N < 10**d	F(w-4).0,4X
Otherwise	sEw.d

Note that the effect of the scale factor is suspended unless the magnitude of the datum to be converted is outside of the range that permits effective use of F conversion.

Note that the effect of the scale factor is suspended unless the magnitude of the datum to be converted is outside of the range that permits effective use of F conversion.

5.13.3.2.3 Complex Conversion

Since a complex datum consists of a pair of separate real data, the conversion is specified by two successively interpreted F, E, G, or D field descritprs. The first of the pair supplies the real part; the second supplies the imaginary part. The two field descriptors may be different.

5.13.3.3 Logical Conversion

The logical field descriptor Lw indicates that the external field occupies w positions and that the list appears, or is to appear, internally as a logical datum.

5.13.3.3b
The external input field must consist of optional blanks followed by a T for true or F for false. The T or F may be followed by optional characters.

5.13.3.3c
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.

5.13.3.4 Character Field Descriptors

Character data may be transmitted by means of two field descriptors, the apostrophe field descriptor and the A[w] field descriptor.

(1) The apsotrophe field descriptor is a character string enclosed in apostrophes. It causes characters to be read into, or written from, the enclosed characters (including blanks) in the format specification itself. Each apostrophe within the character string must be written in the program as two consecutive apostrophes (with no intervening blanks) but must be represented internally by a single apostrophe. An apostrophe in input data to be read into a format specification must be represented by one apostrophe.

(2) The A[w] field descriptor causes characters to be read into, or written from, a specified list item of type character. The list item appears, or is to appear, internally as a character datum. Note that the Ah field descriptor may also be used for Hollerith information when the list item is not of type character.

5.13.3.4b
If a field width w is specified with the A field descriptor, the external field consists of w characters. If the field width w is not specified with the A field descriptor, the length len of the character list item is used as the field width and the external field

field consists of len characters.

5.13.3.4c
Let len be the length of a character list item. If the specified field width w for A input is greater than or equal to len, the rightmost len characters will be taken from the external input field. If the specified field width is less than len, the w characters will appear left-justified with len-w trailing blanks in the internal representation.

5.13.3.4d
If the specified field width w for A output is greater than len, the external output field will consist of w-len blanks followed by the len characters from the internal representation. If the specified field is less than or equal to len, the external output field will consist of the leftmost w characters from the internal representation.

5.13.3.5 Hollerith Field Descriptors

Hollerith information may be transmitted by means of two field descriptors, nH, and Aw.

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

(2) The Aw field descriptor causes w Hollerith characters to be read into, or written from, a specified list item.

5.13.3.5b
Let a be the maximum number of characters that can be stored in a single storage unit at one time. If the field width specified for A input is greater than or equal to g, the rightmost a character will be taken from the external input field. If the field width is less than a, the w characters will appear left-justified with a-w trailing blanks in the internal representation.

5.13.3.5c
If the field width specified for A output is greater than a, the external output field will consist of w-a blanks, followed by the a characters from the internal representation. If the field width is less than or equal to a, the external output field will consist of the leftmost w characters from the internal representation.

5.13.3.5 X Field Descriptor

The nX field descriptor causes n characters to be skipped from the current position. If n is positive, skipping is in the forward direction. If n is negative, skipping is in the backward direction. If a backward skip would result in a zero or negative character position within the record, the effect is a skip to the first character position of the record.

5.13.3.6b
On input, a skip beyond the end of a record is permitted if no characters are transmitted from such character positions.

5.13.3.6c
On output, a skip over character positions that have not previously been filled results in those positions being filled with the character blank. The result is as though the entire record were initially filled with blank characters. If the X field descriptor causes a skip to position c, it causes the length of that output record to be at least c-1 characters.

5.13.3.7 T Field Descriptor

The Tc field descriptor indicates that the transmission of the next character from or to a record is to occur at the cth character position. The first character position of a record is position one. The T field descriptor can be used to move forward or backward from the current position in a record. Input fields can be reread and output fields can be replaced with new characters by using the T field descriptor. For input records, a move beyond the end of a record is permitted if no characters are transmitted from such character positions. For output records, the result is as though the entire record were initially filled with blank characters. The T field descriptor causes the length of the output record to be at least c-1 characters.

5.13.3.8 S Field Descriptor

The S field descriptor may be used to control optional plus signs in numeric external output fields. At the beginning of execution of each formatted output statement, the processor has the option of producing optional plus signs in numeric output fields. If a +S field descriptor is encountered in a format specification, the processor must produce a plus sign in any subsequent position that normally contains an optional plus sign. If a -S field descriptor is encountered, the processor must not produce a plus sign in any subsequent position that normally contains an optional plus sign. If an S field descriptor without a preceding sign is encountered, the option of producing optional plus signs is restored to the processor.

5.13.3.8b
The S field descriptor has no effect during the execution of input statements.

5.13.4 Interaction Between I/O List and Format

The beginning of execution of a formatted READ or formatted WRITE statement initiates format control. Each action of format control depends on information jointly provided (1) by the next field descriptor or field separator obtained from the format specification and (2) by the next item in the input/output list, at least one field descriptor of F, E, G, D, I, L, or A must exist in the format spe-

cification.

5.13.4b
Except for repetition of groups, the format specification is interpreted from left to right.

5.13.4c
The next record is read (1) at the beginning of execution of a formatted READ statement, (2) at each slash encountered under format control, and (3) when format control encounters the end of the format specification before all items in the input list have been assigned values. The joint action of a READ statement and a format specification may not require more characters of a record than the record contains.

5.13.4d
A next record is written (1) at each slash encountered under format control, (2) when the end of the format specification is encountered with more data remaining to be written, and (3) at the completion of execution of the formatted output statement. A record is written at these occurrences even if no characters have been transmitted to the record.

5.13.4e
To each I, F, E, G, D, A, or L basic field descriptor interpreted in a format specification, there corresponds one item specified by the input/output list, except that a complex item requires the interpretation of two F, E, D, or G basic field descriptors. To each H, X,, T, S, or character field descriptor, there is no corresponding item specified by the input/output list, and 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 whenever the next record is read.

5.13.4f
Whenever format control encounters an I, F, E, G, D, A, or L basic field descriptor in a format specification, it determines if there is a corresponding item specified by the input/output list. If there is such an item, it transmits appropriately converted information between the item and the record and then format control proceeds. If there is no corresponding item, format control terminates.

5.13.4g
Whenever format control encounteres a colon, it determines if there are any items remaining in the input/output list. If no items remain format control terminates. If items remain, format control proceeds to the next field descriptor.

5.13.4h
If format control proceeds to the rightmost parenthesis of the format specification, a test is made to determine if another list item is specified. If another list item is not specified, format control terminates. However, if another list item is specified, format control demands that a new record start and format control reverts to the be-

ginning of that optionally repeated format group terminated by the last preceding right parenthesis, or, if non exists, then format control reverts to the first left parenthesis of the format specification. If format control reverts back from the rightmost parenthesis, the reused portion of the format specification must contain at least one F, E, G, D, I, L, or A field descriptor. Note that reversion of format control, of itself, has no effect on the scale factor.

5.14 PROGRAM PUSH, AND PULL STATEMENTS

A PROGRAM statement is of the form:

PROGRAM pgm

where pgm is the symbolic name of the main program in which the PROGRAM statement appears.

5.14b
If this statement appears in an executable program, it must be the first statement of the main program. It is not required to appear. The symbolic name pgm must not be the same as the name of any entity within the main program and it must not be the same as the name of any external procedure, entry, block data subprogram, or common block, in the same executable program.

5.14c
A main program may not be referenced from a subprogram or from itself.

5.14.2 A PUSH statement is of the form:

PUSH (stacknam, exp [,ERR = S])

where stacknam is a stack defined in a STACK statement
exp is an expression that is the same type as stacknam or can be converted to it.

S is a statement label for the error specification if the stack overflows.

5.14.2b
The first argument, stacknam, is a push-down stack and the PUSH statement is the manner in which an entry (exp) is loaded into the stack.

5.15 DEFINITION AND REFERENCE OF SUBROUTINES AND FUNCTIONS

5.15.1 Introduction

There are four categories of procedures:

(1) statement functions
(2) intrinsic functions

(3) external functions
(4) subroutines

5.15.1b
Statement functions, intrinsic functions, and external functions are referred to collectively as functions.

5.15.1c
There are two categories of external functions:

(deleted)
(1) function subprograms
(2) external functions defined by some other means

5.15.1e
Subroutines and external functions are referred to collectively as external procedures.

5.15.1f
If an executable program contains a function or subroutine subprogram, tha subprogram may be referenced within any other program unit of that executable program. A subprogram or main program may not reference itself either directly or indirectly. If an executable program contains an external procedure defined by some means other than as a subprogram, that external procedure may be referenced within any program unit of that executable program. Intrinsic functions may be referenced in any program uni of an executable program except a block data subprogram. A statement function may be referenced only in the program unit that contains the statement function definition.

5.15.1g
Type rules for the names of functions are given in (4.1.2).

5.15.5 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, a logical, or a character assignment statement.

5.15.2b
A symbolic name is a statement function name in a program unit if and only if it meets all three of the following conditions:

(1) A function defining statement (15.17) is present for that symbolic nam
(2) Every appearance of the name, except in a type-statement, is immediately followed by a left parenthesis.
(3) It does not appear in an EXTERNAL statement nor in an array declarator

5.15.3 Intrinsic Functions

The symbolic names of the intrinsic functions (Table 3) are predefined by the processor and have a special meaning and type.

5.15.3b
A symbolic name is an intrinsic function name in a program if and only

if it meets all four of the following conditions:

(1) The name appears in the Symbolic Name column of Table 3.

(2) It does not appear in an EXTERNAL statement nor is it an array name, a character variable name, a subroutine name, or a statement-function name.

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

external procedures.

5.15.1f
If an executable program contains a function or subroutine subprogram, that subprogram may be referenced within any other program unit of that executable program. A subprogram or main program may not reference itself either directly or indirectly. If an executable program contains an external procedure defined by some means other than as a subprogram, that external procedure may be referenced within any program unit of that executable program. Intrinsic functions may be referenced in any program unit of an executable program except a block data subprogram. A statement function may be referenced only in the program unit that contains the statement function definition.

5.15.1g
Type rules for the names of functions are given in (4.1.2).

5.15.5 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, a logical, or a character assignment statement.

5.15.2b
A symbolic name is a statement function name in a program unit if and only if it meets all three of the following conditions:

(1) A function defining statement (15.17) is present for that symbolic name.

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

(3) It does not appear in an EXTERNAL statement nor in an array declarator.

5.15.3 Intrinsic Functions

The symbolic names of the intrinsic functions (Table 3) are predefined by the processor and have a special meaning and type.

5.15.3b
A symbolic name is an intrinsic function name in a program unit if and only if it meets all four of the following conditions:

(1) The name appears in the Symbolic Name column of Table 3.

(2) It does not appear in an EXTERNAL statement nor is it an array name, a character variable name, a subroutine name, or a statement function name.

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

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

The use of an intrinsic function in a program 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.

5.15.3c
Intrinsic functions that cause conversion of an entity from one type to another type provide the same effect as the implied type conversion in assignment statements (Table 2).

5.15.4 External Functions

An external function is defined externally to the program unit that references it.

5.15.4b
There are two kinds of external functions: function subprograms and functions defined by some means other than FORTRAN.

5.15.4c
A function subprogram is an external function that is defined by FORTRAN statements and is headed by a FUNCTION statement.

5.15.4d
A symbolic name is an external function name if it:

(1) appears immediately following the word FUNCTION in a FUNCTION statement, or

(2) appears immediately following the word ENTRY in an ENTRY statement within a function subprogram, or

(3) is not an array name, a character variable name, a statement function name, an intrinsic function name, or a subroutine name and appears immediately followed by a left parenthesis on every occurrence except in a type-statement, in an EXTERNAL statement, as an actual argument, or as a dummy argument in a FUNCTION, a SUBROUTINE, or an ENTRY statement. There must be at least one such appearance in the program unit in which it is so used.

5.15.5 Generic Functions

Table 5 specifies a list of generic function names and the permissible types of arguments and types of results. For those generic functions that require more than one argument, all arguments must be of the same type.

Table 5
GENERIC FUNCTIONS

Symbolic Name	Type of Argument	Type of Result
ABS	Integer Real Double Complex	Integer Real Double Real
INT	Real Double	Integer Integer
NINT	Integer Real Double	Integer Integer Integer
AINT and ANINT	Real Double	Real Double
MOD, MAX, MIN, SIGN, and DIM	Integer Real Double	Integer Real Double
EXP, LOG, SIN, COS, and SORT	Real Double Complex	Real Double Complex
LOG10, TANH, ATAN, ATAN2, ASIN, ACOS, SINH, COSH, and TAN	Real Double	Real Double

5.15.5b
Some of the intrinsic function names are also generic function names that can be used with several different types of arguments and, in most cases, the result of the function is the same type as the actual argument. MAX and MIN are generic function names for choosing the largest value and choosing the smallest value respectively, but are

not the names of any specific intrinsic functions. LOG and LOG10 are generic names for the natural logarithm and the common logarithm respectively, but are not the names of any specific basic external functions. NINT and ANINT are generic names for the nearest integer function; there are no specific function names for the nesrest integer function. The other generic functions have the same mathematical definition as specified in Table 34 for the intrinsic function of the same name.

5.15.5c
If a symbolic name in Table 5 appears in a type-statement within a program unit, that name loses its automatic typing property in that program unit. If that symbolic name appears in Table 3, it can still be used to reference a specific intrinsic function if the four conditions specified in 15.3 are met. (deleted) (deleted) (deleted) (deleted)

5.15.5d
A name in an EXTERNAL statement must be the name of a specific external procedure; it must not be a generic function name that is not defined as a specific external procedure also.

5.15.6 Subroutines

A subroutine is defined externally to the program unit that references it. A subroutine defined by FORTRAN statements and headed by a SUBROUTINE statement is called a subroutine subprogram.

5.15.6b
A symbolic name is a subroutine name if it appears:

(1) immediately following the word SUBROUTINE in a SUBROUTINE statement, or

(2) immediately following the word ENTRY in an ENTRY statement within a subroutine subprogram, or

(3) immediately following the word CALL in a CALL statement.

5.15.7 Dummy and Actual Arguments

Dummy and actual arguments provide a menas of communication between procedures or between a main program and a procedure.

5.15.7.1 Dummy Arguments

Function subprograms, subroutine subprograms, and statement functions use dummy arguments to indicate the types of actual arguments and whether the actual arguments will be variables (or array elements), arrays, subroutines, or external functions. Each dummy argument must be used within a function subprogram or subroutine subprogram as though it is either a variable, an array, a subroutine, or an external function.

5.15.7.1b
At the execution of a function or subroutine reference, an association is established between the corresponding dummy arguments and actual arguments. The first dummy argument becomes associated with the first actual argument, the second dummy argument becomes associated with the second actual argument, etc. All appearances of a dummy argument within a function or subroutine become associated with the actual argument when the function or subroutine is referenced. Except when the actual argument is a subroutine name or a Hollerith constant, a valid association occurs only if the type of the actual argument is the same as the type of the corresponding dummy argument. A Hollerith constant must not become associated with a character dummy argument. Argument association can be carried through more than one level or procedure reference. A valid association exists at the last level only if a valid association exists at all intermediate levels. Argument association within a program unit terminates at the execution of a RETURN or END statement in that program unit.

5.15.7.1c
The number of dummy arguments in a procedure must be the same as the number of actual arguments in each reference to that procedure or procedure entry.

5.15.7.1d
Dummy argument names may appear wherever an actual name of the same class and type could appear except where they are explicitly prohitited. They are not allowed in EQUIVALENCE, DATA, PARAMETER, SAVE, or COMMON statements except as a common block name. A dummy argument name cannot appear as the entry name in an ENTRY statement. Integer dummy arguments may also appear in adjustable dimensions in dummy array declarators. Although dummy arguments are not actual variables, arrays, etc., each dummy argument is herein considered to be either a variable, array, subroutine, or external function.

5.15.7.1e
If a dummy argumebt is a variable, the associated actual argument must be a variable, an array element, an expression, or a Hollerith constant.

5.15.7.1f
If a dummy argument is an array, the associated actual argument must be either an array or an array element. If the actual argument is an array, the length of the dummy argument array must be no greater than the length of the actual argument array and each actual argument array element becomes associated with the dummy argument array element that has the same subscript value as the actual argument array element.

5.15.7.1g
If the actual argument is an array element, 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. When an actual argument is an array element with a subscript value of p, the dummy argument array element with a subscript value of q becomes associated with the actual argument array element that has a subscript value of p+q-1 (Table 1).

Within a program unit, the array declarator given for an array provides all array declarator information needed for that array in an execution of that program unit. The number and size of dimensions in an actual argument array declarator may be different from the number and size of the dimensions in an associated dummy argument array declarator.

5.15.7.1h
If a dummy argument is a subroutine, the associated actual argument must be a subroutine.

5.15.7.1i
If a dummy argument is an external function, the associated actual argument must be an external function. A dummy argument that becomes associated with an external function never has any automatic typing property, even if the dummy argument name appears in Table 5. Therefore, the type of the dummy argument must agree with the type of the result of all specific actual arguments that become associated with the dummy argument. Thus, if a dummy argument name is used as an external function and that name also appears in Table 3, 4, or 5, the processor-defined function corresponding to the dummy argument name is not available for referencing within the subprogram.

5.15.7.1j
If a dummy argument that is an array or a variable becomes defined in a referenced subprogram, the associated actual argument must be a variable, an array element, or an array.

5.15.7.1l
If a dummy argument is of type character, the associated actual argument must be of type character and the length of the dummy argument must be less than or equal to the length of the actual argument. If the character dummy argument is an array, the restriction on length is for the entire array and not for each array element. When the length of a character dummy argument is less than the length len of an associated actual argument, the leftmost len characters of the actual argument are associated with the dummy argument.

5.15.7.1n
If a subprogram reference causes a dummy argument in the referenced subprogram to become associated with another dummy argument in the referenced subprogram, a definition of either dummy argument during execution of that subprogram is prohibited. For example, if a subroutine is headed by

```
SUBROUTINE X (A,B)
```

and is referenced by

```
CALL X (C,C)
```

then the dummy arguments A and B each become associated with the same actual argument C and therefore with each other. The above rule means that neither A nor B may become defined during execution of subroutine X.

5.15.7.11
If a subprogram reference causes a dummy argument to become associated with an entity in a common block in the referenced subprogram, a definition within the subprogram of either the dummy argument or the entity in the common block is prohibited. For example, if a subroutine contains the statements:

```
SUBROUTINE X (A)
COMMON C
```

and is referenced by a program unit that contains the statements:

```
COMMON B
CALL X (B)
```

then the dummy argument A becomes associated with the actual argument B which is associated with C which is in common. The above rule means that neither A nor C may become defined during execution of the subroutine X.

5.15.7.2 Actual Arguments

Actual arguments appear in CALL statements for subroutine references and in expressions for function references. Actual arguments specify the variables, array elements, arrays, subroutines, and external functions that are to be associated with the dummy arguments for a particular reference of a subroutine or function. Actual arguments may be constants and expressions if the associated dummy argument is a variable that is not defined during execution of the referenced external procedure.

5.15.7.2b
The number of actual arguments must be the same as the number of dummy arguments in the procedure or entry in the procedure referenced. The type of each actual argument must agree with the type of its associated dummy argument except when the actual argument is a Hollerith constant or a subroutine name.

5.15.7.2c
If an actual argument is a constant, a variable, or an expression, the associated dummy argument must be a variable.

5.15.7.2e
If an actual argument is an array element, the associated dummy argument must be either a variable or an array.

5.15.7.2f
If an actual argument is an external procedure, the associated dummy argument must be an external procedure. If the associated dummy argument appears in a type-statement or is referenced as a function, the actual argument must be a function. If the dummy argument is referenced as a subroutine, the actual argument must be a subroutine and cannot be typed or referenced as a function.

5.15.7.2g
Note that it may not be possible to determine in a given program unit

5.15.7.11
If a subprogram reference causes a dummy argument to become associated with an entity in a common block in the referenced subprogram, a definition within the subprogram of either the dummy argument or the entity in the common block is prohibited. For example, if a subroutine contains the statements:

```
SUBROUTINE X (A)
COMMON C
```

and is referenced by a program unit that contains the statements:

```
COMMON B
CALL X (B)
```

then the dummy argument A becomes associated with the actual argument B which is associated with C which is in common. The above rule means that neither A nor C may become defined during execution of the subroutine X.

5.15.7.2 Actual Arguments

Actual arguments appear in CALL statements for subroutine references and in expressions for function references. Actual arguments specify the variables, array elements, arrays, subroutines, and external functions that are to be associated with the dummy arguments for a particular reference of a subroutine or function. Actual arguments may be constants and expressions if the associated dummy argument is a variable that is not defined during execution of the referenced external procedure.

5.15.7.2b
The number of actual arguments must be the same as the number of dummy arguments in the procedure or entry in the procedure referenced. The type of each actual argument must agree with the type of its associated dummy argument except when the actual argument is a Hollerith constant or a subroutine name.

5.15.7.2c
If an actual argument is a constant, a variable, or an expression, the associated dummy argument must be a variable.

5.15.7.2e
If an actual argument is an array element, the associated dummy argument must be either a variable or an array.

5.15.7.2f
If an actual argument is an external procedure, the associated dummy argument must be an external procedure. If the associated dummy argument appears in a type-statement or is referenced as a function, the actual argument must be a function. If the dummy argument is referenced as a subroutine, the actual argument must be a subroutine and cannot be typed or referenced as a function.

5.15.7.2g
Note that it may not be possible to determine in a given program unit

whether an argument is a function or a subroutine. If the external procedure name appears only in an EXTERNAL statement and in an actual or dummy argument list, it is not possible to determine whether the symbolic name is a subroutine or function by examination of that program unit alone.

5.15.7.2h
An external procedure must be defined at the time it is used as an actual argument in a reference to another procedure. Intrinsic functions and statement functions must not be used as actual arguments.

5.15.7.2i
If an actual argument is a constant or an expression, the associated dummy argument must not be defined within the subprogram.

5.15.7.2j
If an actual argument is an array element name, its subscript is evaluated just before the association of arguments takes place. Note that the value of the subscript remains constant as long as that association of arguments persists, even if the subscript contains variables whose values change during the association.

5.15.7.2k
If an actual argument becomes associated with a dummy argument that appears in an adjustable dimension or appears in an adjustable length in a character type-statement, that actual argument must be defined as an integer value at the time the procedure is referenced.

5.15.7.2l
The number of characters in an actual argument of type character must be greater than or equal to the number of characters in the associated dummy argument.

5.15.8 Common Storage Areas

The COMMON statement provides an alternate means of communication between procedures or between a main program and a procedure. The variables and arrays in a common block may be defined and referenced in all subprograms that contain a declaration of that common block. Because association is by storage units rather than by name, the names of the variables and arrays may be different in the different subprograms. A reference to a datum in a common block is proper if that datum is in a defined state of the same type as the type of the name used to reference that datum. Either part of a complex datum can also be referenced as a real datum. A Hollerith constant can be referenced by a name of any type, except character. No other differences in type between definition and reference are permitted. Integer variables that have been assigned to statement labels cannot be referenced in any program unit other than the one in which they were assigned (10.3).

5.15.8b
In a subprogram that has declared a named common block, the entities in that block remain defined after the execution of a RETURN or END statement if a common block of the same name has been declared in any program unit that is currently referencing the subprogram either directly or

indirectly. Otherwise, such entities become undefined at the execution of a RETURN or END statement except for those that were initially defined and have neither been subsequently defined nor undefined and those that are specified by SAVE statements. Execution of a RETURN or END statement does not cause undefinition of entities in blank common or in any named common block that appears in the main program.

5.15.8c
Note that common blocks may also be used to reduce the total number of storage units or character storage units required for an executable program by causing two or more subprograms to share some of the same storage units or character storage units. This sharing is permitted if the rules for the definition and referencing of data are not violated.

5.15.9 CALL Statement and Subroutine Reference

A CALL statement is used to reference a subroutine.

5.15.9b
A CALL statement is of the form:

CALL sub [(a [,a] ...)]

where: sub is the symbolic name of a subroutine or an entry in a subroutine.

a is an actual argument.

5.15.9c
Execution of a CALL statement references the subroutine designated by sub. Return of control from the referenced subroutine completes exexution of the CALL statement.

5.15.9d
A subroutine is referenced by a CALL statement. The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments in the defining subprogram. The use of a Hollerith constant or a subroutine name as an actual argument are exceptions to the rule requiring agreement of type. An actual argument in a subroutine reference must be one of the following:

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

5.15.9e
When a CALL statement is executed, the referenced subroutine must be available to the program unit in which the CALL statement appears.

5.15.10 Function Reference

A function reference is of one of the forms:

fun(a [,a]...)

fun()

where: fun is the symbolic name of a function or an entry in a function

a is an actual argument

5.15.10b
Execution of a function reference in an expression references the function designated ty fun. Note that function references appear only in expressions.

5.15.10c
The type of the result of a function reference is the same as the type of the function or entry name. However, the type of the result of a generic function is specified in Table 5. Note that the type of a generic function usually depends upon the type of one or more of its arguments.

5.15.10.1 Referencing External Functions

An external function is referenced by using its reference as a primary in an arithmetic, a logical, or a character expression. Execution of an external function reference results in an association of actual arguments with the corresponding dummy arguments in the referenced function. The resultant value is then made available to the expression that contained the function reference.

5.15.10.1b
The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments in the referenced function. An actual argument in an external function reference must be one of the following:

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

5.15.10.1c
When referencing an external function subprogram, the type of the referencing name must agree with the type declared for that name in the function subprogram; and for character functions, the length must also agree.

5.15.10.1d
Arguments for which the result of an intrinsic function is not mathematically defined or exceeds the numeric range of a processor causes

the result of the function to be undefined. The result of the basic external functions ALOG, DLOG, ALOG10, and DLOG10 is undefined for zero or negative arguments. The result of CLOG is undefined for the complex argument (0.,0.). The result of SORT and DSORT is undefined for negative arguments and the result of DMOD is undefined when the second argument is zero. The result of ASIN, DASIN, ACOS, and DACOS is undefined for arguments whose absolute value is greater than one.

15.10.2 Referencing Statement Functions

A statement function is referenced by using its reference as a primary in an arithmetic, a logical, or a character expression. Execution of a statement function reference results in an association of actual argumentw with the corresponding dummy arguments in the expression of the function definition, and an evaluation of the expression. The resultant value is then made available to the expression that contained the function reference.

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 must be one of the following:

(1) A variable name
(2) An array element name
(3) Any other expression except a Hollerith constant.

5.15.10.3 Referencing Intrinsic Functions

An intrinsic function is referenced by using its reference (15.10) as a primary in an arithmetic or a logical expression. Execution of an intrinsic function reference results in the actions specified in Table 3 based on the values of the actual arguments. The resultant value is then made available to the expression that contained the function reference.

The actual arguments, which constitute the argument list, must agree in type, number, and order with the specification in Table 3 or in Table 5 and may be any expression of the specified type. The intrinsic functions AMOD, MOD, SIGN, ISIGN, and DSIGN are not defined when the value of the second argument is zero. If the first argument of SIGN, ISIGN, or DSIGN is zero, the result is always zero and it is neither positive nor negative.

5.15.11 RETURN Statement

A RETURN statement causes return of control to the referencing program unit.

A RETURN statement is of the form:

```
RETURN
```

5.15.11c
Execution of a RETURN statement terminates the reference of a procedure subprogram. This statement may only appear in a function subprogram or a routine subprogram. Such subprograms may contain more than one RETURN statement. A RETURN statement need not appear in a subprogram.

5.15.11d
Execution of this statement in a subroutine subprogram causes return of control to the current referencing program unit.

5.15.11e
Execution of this statement in a function subprogram causes return of control to the current referencing program unit. The value of the function must be defined and that value is then made available to the referencing program unit.

5.15.11f
If an initially defined entity is in a subprogram and is not in a named common block, the completion of execution of a RETURN statement in that subprogram causes all such entities and their associates at that time (except for initially defined entities that have neither been subsequently defined nor undefined) to become undefined. In this respect, it should be noted that the association between dummy arguments and actual arguments is terminated at the beginning of execution of the RETURN statement.

5.15.11g
If a subprogram contains a named 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 neither been subsequently defined nor undefined and for entities that are specified by SAVE statements. Note that if a named common block appears in a main program, then its entities do not become undefined at the execution of a RETURN statement in subprograms that contain the same named common block.

5.15.11h
Again, it should be emphasized, the redefinition of an initially defined entity in a subprogram sometimes results in an undefinition of that entity at the execution of a RETURN statement.

5.15.11i
Execution of an END statement in a subprogram has the same effect as execution a RETURN statement in that subprogram.

5.15.11j
In the execution of an executable program, a procedure subprogram may not be referenced twice without the execution of a RETURN or END statement in that procedure having intervened.

5.15.12 SUBROUTINE Statement

A SUBROUTINE statement is of the form:

SUBROUTINE sub [(d [,d]...)]

where: sub is the symbolic name of the subroutine to be defined.

d , called a dummy argument, is either a variable name, an array name, or an external procedure name.

5.15.13 Subroutine Subprogram Restrictions

Subroutine subprograms are constructed as specified in 3.5 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 EQUIVALENCE, COMMON, PARAMETER, SAVE, or DATA statements in the subprogram.

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

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

5.15.14 FUNCTION Statement

A FUNCTION statement is of one of the forms:

[typ] FUNCTION fun [()]

[typ] FUNCTION fun (d [,d]...)

where: typ is either INTEGER, INTEGER*2, REAL, DOUBLE PRECISION, COMPLES, LOGICAL, CHARACTER (*len , or BIT

len is the length of the result of the character function

fun is the symbolic name of the function to be defined

d , called a dummy argument, is either a variable name, an array name, or an external procedure name.

5.15.15 Function Subprogram Restrictions

Function subprograms are constructed as specified in 5.3.5 with the following restrictions:

(1) The symbolic name of the function or an associated entry name 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

this variable at the time of execution of any RETURN or END statement in this subprogram is 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 or in a type-statement.

(3) The symbolic names of the dummy arguments may not appear in EQUIVALENCE, COMMON, PARAMETER, SAVE, or DATA statements in the function subprogram.

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

(5) The function subprogram may contain any statements except BLOCK DATA, SUBROUTINE, PROGRAM, another FUNCTION statement, or any statement that directly or indirectly references the function being defined.

5.15.16 ENTRY Statement

An ENTRY statement permits a reference of a subprogram to start with any executable statement within that subprogram.

5.15.16b
An ENTRY statement is of one of the forms:

ENTRY en [( )]

ENTRY en (d [,d] ...)

where: en is the entry name

d , called a dummy argument, is either a variable name, an array name, or an external procedure name.

5.15.16c
The entry name is available for reference in all other program units of an executable program, but not in the program unit that contains the entry name in an ENTRY statement. Entry names appearing in ENTRY statements within subroutine subprograms must be referenced as subroutines and entry names appearing in ENTRY statements within function subprograms must be referenced as external functions. An entry name in an ENTRY statement within a subroutine is in the same class as a subroutine and an entry name in an ENTRY statement within a function is in the same class as an external function. A function entry name can appear in a type-statement and any entry name can appear in an EXTERNAL statement and can be used as an actual argument. The entry name in an ENTRY statement cannot be a dummy argument.

5.15.16d
The ENTRY statement is nonexecutable and may appear anywhere within a function or subroutine subprogram after the FUNCTION or SUBROUTINE statement, except within the range of a DO-loop. When an entry name is used to reference a subprogram, execution of that subprogram begins with

the first executable statement that follows the ENTRY statement in which the entry name appears. The ENTRY statement by itself has no effect on the normal execution sequence. An ENTRY statement may immediately precede an END statement which implies a return to the referencing program unit.

5.15.16e
The order, number, type, and names of the dummy arguments in an ENTRY statement may be different from the order, number, type, and names of the dummy arguments in the FUNCTION statement, SUBROUTINE statement, and other ENTRY statements in the same subprogram. However, each reference to a function, subroutine, or entry must use an actual argument list that a-rees in order, number, and type with the dummy argument list in the corresponding FUNCTION, SUBROUTINE, or ENTRY statement.

5.15.16f
A dummy argument is undefined if it is not currently associated with an actual argument. An adjustable array is undefined if the dummy argument array is not currently associated with an actual argument array or if any of the variables appearing in the adjustable array declarator are not currently associated with an actual argument or are not in a common block. Note that there is no retention of argument association between one reference of a subprogram and the next reference of that subprogram.

5.15.16g
In a subprogram, a dummy argument name must not appear in an executable statement unless it has appeared previously in a SUBROUTINE, FUNCTION, or ENTRY statement.

5.15.16h
All entry names within a function subprogram become associated with the name of the function subprogram whenever that function subprogram is referenced by any of its entry names or by its function name. Therefore, definition of any entry name or the name of the function subprogram cuases definition of all the associated names that are of the same type and causes undefinition of all associated names that are of different type. The function and entry names are not required to be the same type, but at the execution of a RETURN or END statement, the name used to reference the function subprogram must be defined. Note that an entry name may not appear in executable statements that precede the appearance of the entry name in an ENTRY statement.

5.15.17 Definition of Statement Functions

A statement function is defined by one of the forms:

```
fun (d [,d] ...) = e

fun ( ) = e
```

where: fun is the symbolic name of the function

e is an expression

d is a dummy argument

The relationship between fun and e must conform to the assignment rules in 5.10.0 and 5.10.2. The d's, if specified, 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.

5.15.17b
Aside from the dummy arguments, operators, and parentheses, the expression e may contain:

(1) Non-Hollerith constants
(2) Variable and array element references
(3) Intrinsic function references
(4) References to previously defined statement functions
(5) External function references

Each variable reference may be either a reference to a dummy argument of the statement function or a reference to a variable that appears within the same program unit that contains the statement function. The array element references cannot be references to dummy arguments of the statement function.

5.15.17c
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.

5.15.17d
In a program unit, statement function defining statements must appear after the specification statements must appear after the specification statements and before the first executable statement.

5.16 BLOCK DATA SUBPROGRAM

Block data subprograms are used to provide initial values for variables and array elements in named common blocks.

5.16.b
A block data subprogram is nonexecutable. There may be more than one block data subprogram in an executable program.

5.16.1 BLOCK DATA Statements

A BLOCK DATA statement is of the form:

    BLOCK DATA sub

where sub is the symbolic name of the block data subprogram.

5.16.1b
The optional name sub must not be the same as the name of any external procedure, main program, common block, or other block data subprogram in the same executable program. The name sub must not appear as a

symbolic name in any statement in that subprogram except in the BLOCK DATA statement.

5.16.1c
This statement may appear only as the first statement of a block data subprogram.

5.16.2 BLOCK DATA Restrictions

The only statements that may appear in a block data subprogram are BLOCK DATA, IMPLICIT, PARAMETER, DIMENSION, COMMON, EQUIVALENCE, DATA, END, and type-statements. Comment lines may appear anywhere before the last line which must be an END statement (3.5).

5.16.2b
If any entity in a given named common block is provided with an initial value in such a subprogram, a complete set of specification statements for the entire block must be included, even though some of the entities in the block do not appear in DATA statements. More than one named common block may have initial values provided for its entities in a single block data subprogram.

5.16.2c
The same named common block may not be specified in more than one block data subprogram in the same executable program.

INTRINSIC FUNCTIONS

Intrinsic Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function
Absolute Value	$\|a\|$	1	ABS IABS DABS	Real Integer Double	Real Integer Double
Truncation	$[a]$ see Note 1	1	AINT INT INDINT DINT	Real Real Double Double	Real Integer Integer Double
Remaindering	a_1 (modulo a_2)	2	AMOD MOD	Real Integer	Real Integer
Choosing Largest Value	Max $(a_1,a_2,\ldots)$	2	AMAX0 AMAX1 MAX0 MAX1 DMAX1	Integer Real Integer Real Double	Real Real Integer Integer Double
Choosing Smallest Value	Min $(a_1,a_2\ldots)$	2	AMINO AMIN1 MIN0 MIN1 DMIN1	Integer Real Integer Real Double	Real Real Integer Integer Double
Float	Conversion from Integer to Real or Double	1	FLOAT DFLOAT	Integer Integer	Real Double
Fix	Conversion to Integer	1	IFIX	Real	Integer
Transfer of Sign	$\|a_1\|$ if $a_2 \geq 0$ $-\|a_1\|$ if $a_2 < 0$	2	SIGN ISIGN DSIGN	Real Integer Double	Real Integer Double
Positive Difference	a_1 -Min (a_1,a_2)	2	DIM IDIM DDIM	Real Integer Double	Real Integer Double
Single	Conversion from Double to Real	1	SNGL	Double or Real	Real

TABLE 3 (Continued)
INTRINSIC FUNCTIONS

Intrinsic Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function
Length of Character Entity		1	LEN	Character	Integer
Obtain Real Part of Complex Argument	ar	1	REAL	Complex	Real
Obtain Imaginary Part of Complex Argument	ai	1	AIMAG	Complex	Real
Double	Conversion from Real to Double	1	DBLE	Real or Double	Double
Express Two Real Arguments in Complex form	$a_1 + a_2$ /-1	2	CMPLX	Real	Complex
Obtain Conjugate of a Complex Argument	ar - ai / -1	1	CONJG	Complex	Complex
Double Precision Product of two Real Arguments	$a_1 + a_2$	2	DPROD	Real	Double

NOTES FOR TABLE 3:

(1) $\lceil x \rceil$ is defined as the largest integer whose magnitude does not exceed the magnitude of x and whose sign is the same as x.

(2) The remaindering function MOD or AMOD (a_1,a_2) is defined as $a_1 - \lceil a_1/a_2 \rceil a_2$, where $\lceil x \rceil$ is defined in (1) above.

(3) Intrinsic functions that cause converstion of an entity from one type to another type provide the same effect as the implied conversion in assignment statements (Table 2). The functions, SNGL with a real argument, and DBLE with a double precision argument, return the value of the argument without conversion

(4) A complex value is expressed as an ordered pair of reals, (ar,ai) where ar is the real part and ai is the imaginary part.

TABLE 4
BASIC EXTERNAL FUNCTIONS

Basic External	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function	Range of Result
Exponential	e**a	1	EXP DEXP CEXP	Real Double Complex	Real Double Complex	$r \geq 0$
Natural Logarithm	loge(a)	1	ALOG DLOG CLOG	Real Double Complex	Real Double Complex	
Common Logarithm	log10(a)	1	ALOG10 DLOG10	Real Double	Real Double	
Sine	sin(a)	1	SIN DSIN CSIN	Real Double Complex	Real Double Complex	$-1 \leq r \leq 1$
Cosine	cos(a)	1	COS DCOS CCOS	Real Double Complex	Real Double Complex	$-1 \leq r \leq 1$
Tangent	tan(a)	1	TAN DTAN	Real Double	Real Double	
Arctangent	arctan(a)	1	ATAN DATAN	Real Double	Real Double	$-pi/2 \leq r \leq pi/2$
	arctan(a_1/a_2)	2	ATAN2 DATAN2	Real Double	Real Double	$-pi < r \leq pi$
Arcsine	arcsin(a)	1	ASIN DASIN	Real Double	Real Double	$-pi/2 \leq r \leq pi/2$
Arcosine	arcos(a)	1	ACOS DACOS	Real Double	Real Double	$0 \leq r \leq pi$

TABLE 4 (Continued)
BASIC EXTERNAL FUNCTIONS

Basic External	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function	Range of Result
Hyperbolic Sine	sinh(a)	1	SINH DSINH	Real Double	Real Double	
Hyperbolic Cosine	cosh(a)	1	COSH DCOSH	Real Double	Real Double	$r \geq 1$
Hyperbolic Tangent	tanh(a)	1	TANH DTANH	Real Double	Real Double	$-1 < r < 1$
Square Root	$(a)^{1/2}$	1	SORT DSORT CSORT	Real Double Complex	Real Double Complex	r 0 Real part 0
Remaindering	a_1 (modulo a_2)	2	DMOD	Double	Double	
Modulus	$(ar**2+ai**2)^{1/2}$	1	CABS	Complex	Real	r 0

NOTES FOR TABLE 4:

(1) The argument of ALOG, DLOG, ALOG10, and DLOG10 must be greater than zero. The argument of CLOG must not be (o.,0.)

(2) The argument of SIN, DSIN, COS, DCOS, TAN and DTAN must be in radians. These functions use the argument modulo 2 pi.

(3) The argument of SORT and DSORT must be greater than or equal to zero.

(4) The absolute value of the argumnet of ASIN, DASIN, ACOS, and DACOS must be less than or equal to one.

(5) The result of ASIN, DASIN, ACOS, DACOS, ATAN, DATAN, ATAN2, and DATAN2 is in radians.

(6) The result of ATAN2 and DATAN is zero or positive for $a_1 \geq 0$ and negative for $a_1 < 0$. The result is undefined if both arguments are zero.

(7) The result of CSORT is the principal value with the real part greater than or equal to zero. When the real part is zero, the imaginary part is greater than or equal to zero.

(8) The remaindering function DMOD (a_1, a_2) is de-

NOTES FOR TABLE 4 (Continued)

(8) Cont. fined as $a_1 - \lfloor a_1/a_2 \rfloor a_2$, where $\lfloor x \rfloor$ is the largest integer whose magnitude does not exceed the magnitude of x and whose sign is the same as the sign of x.

(9) The principle value is used for results of complex functions.

(10) The argument and result of a function cannot exceed the numerical range of the processor. In particular, if the value of a function can mathematically be infinite, the argument must be such that the result does not exceed the numerical range of the processor.

(11) A complex value is expressed as an ordered pair of reals, (ar,ai), where ar is the real part and ai is the imaginary part.

6. CONFIGURATION REQUIRED

A. Any INTERDATA 32 bit processor.

B. OS/32

C. Source input, source output, a direct access scratch device, and listing output device.

D. Memory requirements.

7. CONFIGURATION OPTIONS

8. RELATION TO OTHER PRODUCTS OR PROGRAMS

FORTRAN V LEVEL II is nearly upwards compatible with FORTRAN V LEVEL I. Hollerith constants, because of the addition of character strings, are only legal in the argument list of a CALL statement and in the DATA statement.

The FORTRAN V LEVEL II compiler will generate CAL source code for 32 bit processors. The code produced is compatible with the OS/32 resident loader, the OS/32 MT TET, and the OS/32 Library Loader.

9. PERFORMANCE SPECIFICATIONS

10. PACKAGING