CONTROL DATA CORPORATION STANDARD FORTRAN

LANGUAGE SPECIFICATION

Submitted by: _______________
E.S.Mast

Approved by: _______________
A.D.Tait

Approved by: _______________
R.M.Hudson

____
A.W.Hartley

DISCLAIMER:

This document is an internal working paper and does not represent any intent on the part of CONTROL DATA CORPORATION.

REVISION RECORD	
REVISION	DESCRIPTION
A 83-06-30	Original version.
B 83-08-30	Resolution of comments from division review

1.0 INTRODUCTION

1.1 PURPOSE

This standard specifies the form and establishes the interpretation of programs expressed in the Control Data Corporation Standard FORTRAN language. The purpose of this standard is to promote portability of FORTRAN programs for use on CDC data processing systems.

1.2 PROCESSOR

The combination of a data processing system and the mechanism by which programs are transformed for use on that data processing system is called a processor in this standard.

1.3 SCOPE

1.3.1 INCLUSIONS.

This standard specifies:

(1) The form of a program written in the CDC Standard FORTRAN language

(2) Rules for interpreting the meaning of such a program and its data

(3) The form of writing input data to be processed by such a program operating on data processing systems

(4) The form of the output data resulting from the use of such a program on data processing systems

1.3.2 EXCLUSIONS.

This standard does not specify:

(1) The mechanism by which programs are transformed for use on a data processing system

(2) The method of transcription of programs or their input or output data to or from a data processing medium

(3) The operations required for setup and control of the use of programs on data processing systems

(4) The results when the rules of this standard fail to establish an interpretation

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

(6) The range or precision of numeric quantities and the method of rounding of numeric results

(7) The physical properties of input/output records, files, and units

(8) The physical properties and implementation of storage

1.4 CONFORMANCE

The requirements, prohibitions, and options specified in this standard generally refer to permissible forms and relationships for standard-conforming programs rather than for processors. The obvious exceptions are the optional output forms produced by a processor, which are not under the control of a program. The requirements, prohibitions, and options for a standard-conforming processor usually must be inferred from those given for programs.

An executable program (2.4.2) conforms to this standard if it uses only those forms and relationships described herein and if the executable program has an interpretation according to this standard. A program unit (2.4) conforms to this standard if it can be included in an executable program in a manner that allows the executable program to be standard conforming.

A processor conforms to this standard if it executes standard-conforming programs in a manner that fulfills the interpretations prescribed herein. A standard-conforming processor may allow additional forms and relationships provided that such additions do not conflict with the standard forms and relationships. However, a standard-conforming processor may allow additional intrinsic functions (16.10) even though this could cause a conflict with the name of an external function in a standard-conforming program. If such a conflict occurs, the processor is permitted to use the intrinsic function unless the name appears in an EXTERNAL statement within the program unit. A standard-conforming program must not use intrinsic functions that have been added by the processor. Note that a standard-conforming program must not use any forms or relationships that are prohibited by this standard, but a standard-conforming processor may allow such forms and relationships if they do not change the proper interpretation of a standard-conforming program.

Because a standard-conforming program may place demands on the processor that are not within the scope of this standard or may include standard items that are not portable, such as external procedures defined by means other than FORTRAN, conformance to this

standard does not ensure that a standard-conforming program will execute consistently on all or any standard-conforming processors.

1.5 NOTATION USED IN THIS STANDARD

In this standard, "must" is to be interpreted as a requirement; conversely, "must not" is to be interpreted as a prohibition.

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

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

(2) Lowercase letters and lowercase words indicate general entities for which specific entities must be substituted in actual statements. Once a given lowercase letter or word is used in a syntactic specification to represent an entity, all subsequent occurrences of that letter or word represent the same entity until that letter or word is used in a subsequent syntactic specification to represent a different entity.

(3) Brackets, [ ], are used to indicate optional items.

(4) An ellipsis, ... , indicates that the preceding optional items may appear one or more times in succession.

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

(6) Words or groups of words that have special significance are underlined where their meaning is described. Titles and the metalanguage symbols described in 1.5(2) above are also underlined.

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

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

the following forms are allowed:

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

1.0 INTRODUCTION
1.5 NOTATION USED IN THIS STANDARD

When an actual statement is written, specific entities are substituted for _sub_ and each _a_; for example:

```
CALL ABCD (X,1.0)
```

2.0 FORTRAN TERMS AND CONCEPTS

This section introduces basic terminology and concepts, some of which are clarified further in later sections. Many terms and concepts of more specialized meaning are also introduced in later sections. The underlined words are described here and used throughout this standard.

2.1 SEQUENCE

A sequence is a set ordered by a one-to-one correspondence with the numbers 1, 2, through n. The number of elements in the sequence is n. A sequence may be empty, in which case it contains no elements.

The elements of a nonempty sequence are referred to as the first element, second element, etc. The nth element, where n is the number of elements in the sequence, is called the last element. An empty sequence has no first or last element.

2.2 SYNTACTIC_ITEMS

Letters, digits, and special characters of the FORTRAN character set (3.1) are used to form the syntactic items of the FORTRAN language. The basic syntactic items of the FORTRAN language are constants, symbolic names, statement labels, keywords, operators, and special characters.

The form of a constant is described in Section 4.

A symbolic name takes the form of a sequence of 1 to 31 letters, digits, or underscores, the first of which must be a letter. Classification of symbolic names and restrictions on their use are described in Section 19.

A statement label takes the form of a sequence of one to five digits, one of which must be nonzero, and is used to identify a statement (3.4).

A keyword takes the form of a specified sequence of letters. The keywords that are significant in the FORTRAN language are described in Sections 7 through 16. In many instances, a keyword or a portion of a keyword also meets the requirements for a symbolic name. Whether a particular sequence of characters identifies a keyword or a symbolic name is implied by context. There is no sequence of characters that is reserved in all contexts in FORTRAN.

The set of special characters is described in 3.1.4. A special character may be an operator or part of a constant or have some

other special meaning. The interpretation is implied by context.

2.3 STATEMENTS, COMMENTS, AND LINES

A FORTRAN *statement* is a sequence of syntactic items, as described in Sections 7 through 16. Except for assignment and statement function statements, each statement begins with a keyword. In this standard, the keyword or keywords that begin the statement are used to identify that statement. For example, a DATA statement begins with the keyword DATA.

A statement is written in one or more lines, the first of which is called an *initial line* (3.2.2); succeeding lines, if any, are called *continuation lines* (3.2.3).

There is also a line called a *comment line* (3.2.1), which is not part of any statement and is intended to provide documentation.

2.3.1 CLASSES OF STATEMENTS.

Each statement is classified as executable or nonexecutable (Section 7). Executable statements specify actions. Nonexecutable statements describe the characteristics, arrangement, and initial values of data; contain editing information; specify statement functions; classify program units; and specify entry points within subprograms.

2.4 PROGRAM UNITS AND PROCEDURES

A *program unit* consists of a sequence of statements and optional comment lines. A program unit is either a main program or a subprogram.

A *main program* is a program unit that does not have a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement; it may have a PROGRAM statement as its first statement.

A *subprogram* is a program unit that has a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement. A subprogram whose first statement is a FUNCTION statement is called a *function subprogram*. A subprogram whose first statement is a SUBROUTINE statement is called a *subroutine subprogram*. Function subprograms and subroutine subprograms are called *procedure subprograms*. A subprogram whose first statement is a BLOCK DATA statement is called a *block data subprogram*.

2.4.1 PROCEDURES.

Subroutines (16.6), external functions (16.5), statement functions (16.4), and the intrinsic functions (16.3) are called *procedures*. Subroutines and external functions are called *external procedures*.

Function subprograms and subroutine subprograms may specify one or more external functions and subroutines, respectively (16.7). External procedures may also be specified by means other than FORTRAN subprograms.

2.4.2 EXECUTABLE PROGRAM.

An executable program is a collection of program units that consists of exactly one main program and any number, including none, of subprograms and external procedures.

2.5 VARIABLE

A variable is an entity that has both a name and a type. A variable name is a symbolic name of a datum. Such a datum may be identified, defined (2.12), and referenced (2.13). Note that the usage in this standard of the word "variable" is more restricted than its normal usage, in that it does not include array elements.

The type of a variable is optionally specified by the appearance of the variable name in a type-statement (8.4). If it is not so specified, the type of a variable is implied by the first letter of the variable name to be integer or real (4.1.2), unless the initial letter type implication is changed by the use of an IMPLICIT statement (8.5).

At any given time during the execution of an executable program, a variable is either defined or undefined (2.12).

2.6 ARRAY

An array is a (possibly empty) sequence of data that has a name and a type. The name of an array is a symbolic name. An array name is the symbolic name of a sequence of data. Such a sequence of data may be identified, defined, and referenced. At any given time during the execution of an executable program, an array is either defined or undefined.

2.6.1 ARRAY ELEMENTS

Each of the elements of an array is called an array element. An array name qualified by a subscript is an array element name and identifies a particular element of the array (5.4), unless the array has size zero. Such a datum may be identified, defined (2.12), and referenced (2.13). The number of array elements in an array is specified by an array declarator (5.2).

An array element has a type. The type of all array elements within an array is the same, and is optionally specified by the appearance of the array name in a type-statement (8.4). If it is not so specified, the type of an array element is implied by the first

letter of the array name to be integer or real (4.1.2), unless the initial letter type implication is changed by the use of an IMPLICIT statement (8.5).

At any given time during the execution of an executable program, an array element is either defined or undefined (2.12).

2.6.2 ARRAY SECTIONS

An array section is a subsequence of an array. An array name qualified by a section subscript identifies a particular subsequence of the array. The type of all array elements within an array section is the same as the type of the array name.

At any given time during the execution of an executable program, an array section is either defined or undefined.

2.7 SUBSTRING

A character datum is a (possibly empty) sequence of characters. A substring is a contiguous portion of a character datum. The form of a substring name used to identify, define (2.12), or reference (2.13) a substring is described in 5.10.1.

At any given time during the execution of an executable program, a substring is either defined or undefined (2.12).

2.8 SCALAR

A scalar is a constant, variable, array element, scalar-valued function reference, or substring. The value of an expression is a scalar if each primary in the expression is a scalar.

2.9 DUMMY ARGUMENT

A dummy argument in a procedure is either a symbolic name or an asterisk. A symbolic name dummy argument identifies a variable, array, or procedure that becomes associated (2.15) with an actual argument of each reference (2.13) to the procedure (16.2, 16.4.2, 16.5.2, and 16.6.2). An asterisk dummy argument indicates that the corresponding actual argument is an alternate return specifier (16.6.2.3, 16.8.3, and 16.9.3.5).

Each dummy argument name that is classified as a variable, array, or dummy procedure may appear wherever an actual name of the same class (Section 19) and type may appear, except where explicitly prohibited.

2.10 SCOPE OF SYMBOLIC NAMES AND STATEMENT LABELS

The scope of a symbolic name (19.1) is an executable program, a program unit, a statement function statement, or an implied-DO list in a DATA statement.

The name of the main program and the names of block data subprograms, external functions, subroutines, and common blocks have a scope of an executable program.

The names of variables, arrays, constants, statement functions, intrinsic functions, dummy procedures, and NAMELIST group names have a scope of a program unit.

The names of variables that appear as dummy arguments in a statement function statement have a scope of that statement.

The names of variables that appear as the DO-variable of an implied-DO in a DATA statement have a scope of the implied-DO list.

Statement labels have a scope of a program unit.

2.11 LIST

A *list* is a nonempty sequence (2.1) of syntactic entities separated by commas. The entities in the list are called *list items*.

2.12 DEFINITION STATUS

At any given time during the execution of an executable program, the *definition status* of each variable, array, array section, array element, or substring is either *defined* or *undefined* (Section 18). A defined entity has a value. The value of a defined entity does not change until the entity becomes undefined or is redefined with a different value. Zero-sized arrays, array sections and substrings are taken always to be defined with a special value that depends on their shape and type.

If a variable, array, array section, array element, or substring is undefined, it does not have a predictable value.

A previously defined variable, non-zero sized array, non-zero sized array section, or array element may become undefined. Subsequent definition of a defined variable, array, array section, or array element is permitted, except where it is explicitly prohibited.

A character entity is defined if it has size zero or if every substring of length one of the entity is defined. Note that if a string is defined, every substring of the string is defined, and if any substring of the string is undefined, the string is undefined. Defining any substring does not cause any other string or substring

to become undefined.

An entity is initially defined if it has size zero or if it is assigned a value in a DATA statement (Section 9). Initially defined entities are in the defined state at the beginning of execution of an executable program. All variables and array elements not initially defined, nor associated (2.15) with an initially defined entity, are undefined at the beginning of execution of an executable program.

An entity must be defined at the time a reference to it is executed.

2.13 REFERENCE

A variable, array, array section, array element, or substring reference is the appearance of a variable, array, array section, array element, or substring name, respectively, in a statement in a context requiring the value of that entity to be used during the execution of the executable program. When a reference to an entity is executed, its current value is available. In this standard, the act of defining an entity is not considered a reference to that entity.

A procedure reference is the appearance of a procedure name in a statement in a context that requires the actions specified by the procedure to be executed during the execution of the executable program. When a procedure reference is executed, the procedure must be available.

2.14 STORAGE

A storage sequence is a sequence of storage units. A storage unit is either a numeric storage unit, a character storage unit, or a bit storage unit.

An integer, real, Boolean, or logical datum has one numeric storage unit in a storage sequence. A double precision or complex datum has two numeric storage units in a storage sequence. A half precision storage unit is half of a numeric storage unit. A character datum has one character storage unit in a storage sequence for each character in the datum. A bit datum has one bit storage unit in a storage sequence for each bit in the datum.

A storage unit has a sequence of consecutive bit positions. A bit position of a storage unit may hold a component of a datum having that storage unit. This component is a bit value, or bit, of 0 or 1. A numeric storage unit has 64 bit positions. A character storage unit has 8 bit positions. A bit storage unit has 1 bit position.

If a datum requires more than one storage unit in a storage sequence, those storage units are consecutive.

The concept of a storage sequence is used to describe relationships that exist among variables, array elements, arrays, substrings, and common blocks. This standard does not specify a relationship between the storage sequence concept and the physical properties or implementation of storage.

2.15 ASSOCIATION

Association of entities exists if the same datum may be identified by different symbolic names in the same program unit, or by the same name or a different name in different program units of the same executable program (18.1).

Entities may become associated by the following:

(1) Common association (8.3.4)

(2) Equivalence association (8.2.2)

(3) Argument association (16.9.3)

(4) Entry association (16.7.3)

2.16 SHAPE

The shape of an array or array section is the number of dimensions, the size of each dimension, and whether the array is defined as rowwise or columnwise.

The shape of an array is denoted by (d1, d2, ..., dn) where n is the number of dimensions and di is the size of the ith dimension.

Two array objects (array, array section, array-valued expression, array-valued function, elemental function reference with an array result) are of the same shape only if they have the same number of dimensions, the size of each dimension is the same, and they are both columnwise or both rowwise.

The shape of a scalar is zero dimensions.

The shape of a user array-valued function is determined by the declaration for the function name as an array in the PROCEDURE INTERFACE INFORMATION BLOCK. The shape of an elemental intrinsic function is the shape of the argument with the greatest number of dimensions.

2.17 CONFORMABILITY

Two array objects (array, array section, array-valued expression, array-valued function, elemental function reference with an array result) are *conformable* if they have the same shape. A scalar is conformable with any array, array section, or array expression. The scalar is treated as if it had been extended to an array with the same shape in which all array elements have the value of the scalar.

2.18 ALLOCATABLE

An allocatable array is *allocated* if it has been specified during the execution of an ALLOCATE statement and not subsequently released. An allocatable array is no longer allocated if it has been specified during the execution of a FREE statement, or if execution of the subprogram in which it was declared has been terminated by execution of a RETURN or END statement and the array name is not specified by a SAVE statement.

3.0 CHARACTERS, LINES, AND EXECUTION SEQUENCE

3.1 FORTRAN CHARACTER SET

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

3.1.1 LETTERS

A *letter* is one of the twenty-six characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Each letter also has a lower case form. Except in a character constant (4.8.1), Hollerith constant (4.9.1.1), apostrophe edit descriptor (14.5.1), quote edit descriptor (14.5.1), or H edit descriptor (14.5.2), the lower case form of a letter is interpreted as merely a variation in the graphic used for the letter. In a character constant, Hollerith constant, apostrophe edit descriptor, quote edit descriptor, or H edit descriptor, the lower case form of a letter is interpreted as a different character from the upper case form of the letter, except as explicitly noted elsewhere within this language specification.

3.1.2 DIGITS.

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 numeric interpretation is appropriate.

An *octal digit* is one of the eight characters:

0 1 2 3 4 5 6 7

The digits in an octal constant (4.9.1.2) must all be octal digits.

A *hexadecimal digit* is one of the sixteen characters:

0 1 2 3 4 5 6 7 8 9 A B C D E F

The digits in a hexadecimal constant (4.9.1.3) must all be hexadecimal digits. Except in an octal or hexadecimal constant, a string of digits is interpreted in the decimal base number system when a number system base interpretation is appropriate. In an octal constant, a string of octal digits is interpreted in the octal base number system. In a hexadecimal constant, a string of

hexadecimal digits is interpreted in the hexadecimal number system.

3.1.3 ALPHANUMERIC CHARACTERS

An *alphanumeric character* is a letter, a digit, or the underscore.

3.1.4 SPECIAL CHARACTERS.

A *special character* is one of the fifteen characters:

Character	Name of Character
	Blank
=	Equals
+	Plus
-	Minus
*	Asterisk
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Decimal Point
$	Currency Symbol
'	Apostrophe
:	Colon
"	Quote
_	Underscore

The set of characters capable of representation in the processor is given in Appendix A.

3.1.5 COLLATING SEQUENCE AND GRAPHICS.

Note: Throughout this section, S denotes the size of the processor's character set. See Appendix A for value of S.

Character relational expressions are evaluated according to a collating sequence, determined by a *collation weight table*. A weight table is a one-dimensional integer array of size S. Each element of the weight table has a value between zero and S-1 inclusive. The value of element i of the weight table is the *collating weight* for the character with the *character code* of i. The character codes and graphic representations for all characters supported by the processor are given in Appendix A. If c(i) and c(j) are characters and i' and j' are their respective collating weights, then

> c(i) .op. c(j) has the value true if and only if i'.op.j' has the value true, where .op. is any of the relational operators.

The value of a weight table element does not have to be unique within the table; i.e., several characters may have the same collating weight.

Collation may be directed by the _fixed collation weight table_ or by the _user-specified collation weight table_. The tables are predefined with the values given in Appendix A. The fixed table may not be modified but the program may access and modify the user-specified table. A compiler-call statement option or the collation control directive (3.7.3) determines which table specifies collation sequence for character relational expressions.

The collating sequence used by the intrinsic functions LGE, LGT, LLE, and LLT (16.10) is independent of both the fixed and user-specified weight tables and is therefore unaffected by either the compiler-call statement option or the collation control directive. The intrinsic function INDEX (16.10) does not use either collation weight table. The intrinsic functions CHAR and ICHAR (16.10) return values dependent upon the collation weight tables.

A program may access or modify the user-specified collation weight table by using the procedures COLSEQ, CSOWN or WTSET (16.12.7, 16.12.9, 16.12.8).

3.1.6 BLANK CHARACTER.

With the exception of the uses specified (3.2.2, 3.2.3, 3.3, 4.8, 4.8.1, 4.9.1.1, 14.5.1, and 14.5.2), a blank character within a program unit has no meaning and may be used to improve the appearance of the program unit, subject to the restriction on the number of consecutive continuation lines (3.3).

3.2 LINES

A _line_ in a program unit is a sequence of 72 characters. All characters must be from the FORTRAN character set, except as described in 3.2.1, 4.8, 4.9.1.1, 13.2.2, and 14.2.1.

The character positions in a line are called _columns_ and are numbered consecutively 1, 2, through 72. The number indicates the sequential position of a character in the line, beginning 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.

3.2.1 COMMENT LINE

A _comment line_ is any line that contains a C or an asterisk in column 1, or contains only blank characters in columns 1 through 72. A comment line that contains a C or an asterisk in column 1 may contain any character capable of representation in the processor in

columns 2 through 72.

A comment line does not affect the executable program in any way and may be used to provide documentation. Certain special forms of comment lines control processor behavior. These comment lines are processor directives called CS-directives (3.7).

Comment lines may appear anywhere in the program unit. Comment lines may precede the initial line of the first statement of any program unit. Comment lines may appear between an initial line and its first continuation line or between two continuation lines.

3.2.2 INITIAL LINE.

An initial line is any line that is not a comment line and contains the character blank or the digit 0 in column 6. Columns 1 through 5 may contain a statement label (3.4), or each of the columns 1 through 5 must contain the character blank.

3.2.3 CONTINUATION LINE.

A continuation line is any line that contains any character of the FORTRAN character set other than the character blank or the digit 0 in column 6 and contains only blank characters in columns 1 through 5. A statement must not have more than nineteen continuation lines.

3.3 STATEMENTS

The statements of the FORTRAN language are described in Sections 7 through 17 and are used to form program units. Each statement is written in columns 7 through 72 of an initial line and as many as nineteen continuation lines. An END statement is written only in columns 7 through 72 of an initial line. No other statement in a program unit may have an initial line that appears to be an END statement. Note that a statement must not contain more than 1320 characters. Except as part of a logical IF statement (11.5) or a logical WHERE statement (11.15), no statement may begin on a line that contains any part of the previous statement.

Blank characters preceding, within, or following a statement do not change the interpretation of the statement, except when they appear within the datum strings of character constants, in Hollerith constants, or the H or apostrophe edit descriptors in FORMAT statements. However, blank characters do count as characters in the limit of total characters allowed in any one statement.

3.4 STATEMENT LABELS

Statement labels provide a means of referring to individual statements. Any statement may be labeled, but only labeled executable statements and FORMAT statements may be referred to by

the use of statement labels. The form of a statement label is a sequence of one to five digits, one of which must be nonzero. The statement label may be placed anywhere in columns 1 through 5 of the initial line of the statement. The same statement label must not be given to more than one statement in a program unit. Blanks and leading zeros are not significant in distinguishing between statement labels.

3.5 ORDER OF STATEMENTS AND LINES

A PROGRAM statement may appear only as the first statement of a main program. The first statement of a subprogram must be either a FUNCTION, SUBROUTINE, or BLOCK DATA statement.

Within a program unit that permits the statements:

(1) FORMAT statements may appear anywhere;

(2) all specification statements must precede all DATA statements, NAMELIST statements, statement function statements, and executable statements;

(3) all statement function statements must precede all executable statements;

(4) DATA statements may appear anywhere after the specification statements;

(5) ENTRY statements may appear anywhere except between a block IF statement and its corresponding END IF statement, between a block WHERE statement and its corresponding END WHERE statement, or between a DO statement and the terminal statement of its DO-loop; and

(6) A NAMELIST statement (13.14) may appear anywhere after the specification statements, but must precede any data transfer input/output statement that refers to the NAMELIST group name (13.14, 19.2.12) that is specified by the NAMELIST statement.

Within the specification statements of a program unit, IMPLICIT statements must precede all other specification statements except PARAMETER statements. Any specification statement that specifies the type of a symbolic name of a constant must precede the PARAMETER statement that defines that particular symbolic name of a constant; the PARAMETER statement must precede all other statements containing the symbolic names of constents that are defined in the PARAMETER statement.

The last line of a program unit must be an END statement.

3.0 CHARACTERS, LINES, AND EXECUTION SEQUENCE
3.5 ORDER OF STATEMENTS AND LINES

Figure 1

Required Order of Statements and Comment Lines

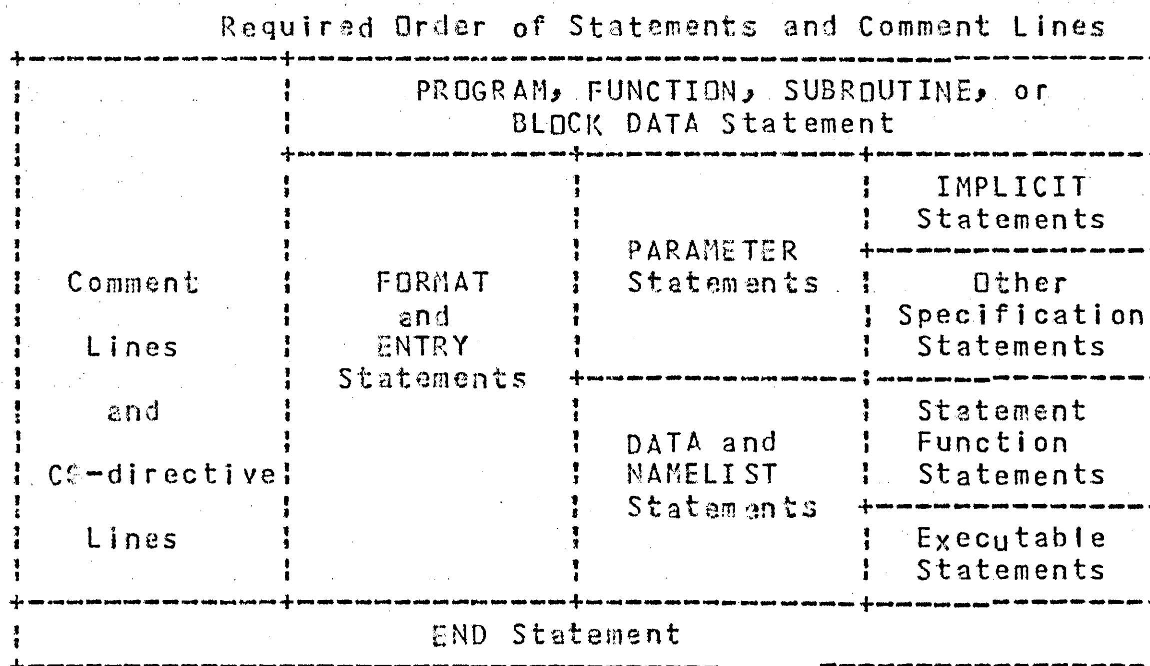

Figure 1 is a diagram of the required order of statements and comment lines for a program unit. Vertical lines delineate varieties of statements that may be interspersed. For example, FORMAT statements may be interspersed with statement function statements and executable statements. Horizontal lines delineate varieties of statements that must not be interspersed. For example, statement function statements must not be interspersed with executable statements. Note that an END statement is also an executable statement and must appear only as the last statement of a program unit.

3.6 NORMAL EXECUTION SEQUENCE AND TRANSFER OF CONTROL

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 an external procedure specified in a subprogram is referenced, execution begins with the first executable statement that follows the FUNCTION, SUBROUTINE, or ENTRY statement that specifies the referenced procedure as the name of a procedure.

A transfer of control is an alteration of the normal execution sequence. Statements that may cause a transfer of control are:

(1) GO TO

(2) Arithmetic IF

(3) RETURN

(4) STOP

(5) An input/output statement containing an error specifier or end-of-file specifier

(6) CALL with an alternate return specifier

(7) A logical IF statement containing any of the above forms

(8) Block IF and ELSE IF

(9) The last statement, if any, of an IF-block or ELSE IF-block

(10) DO

(11) The terminal statement of a DO-loop

(13) END

The effect of these statements on the execution sequence is described in Sections 11, 13, and 16.

The normal execution sequence is not affected by the appearance of nonexecutable statements or comment lines between executable statements. Execution of a function reference or a CALL statement is not considered a transfer of control in the program unit that contains the reference, except when control is returned to a statement identified by an alternate return specifier in a CALL statement. Execution of a RETURN or END statement in a referenced procedure, or execution of a transfer of control within a referenced procedure, is not considered a transfer of control in the program unit that contains the reference.

In the execution of an executable program, a procedure subprogram must not be referenced a second time without the prior execution of a RETURN or END statement in that procedure.

3.7 <u>CS-DIRECTIVES</u>

A <u>CS-directive</u> is a special form of comment line that affects processor behavior. This effect can be suppressed, and the CS-directive treated as a comment line, by selection of the appropriate FORTRAN control statement option (see host operating system reference manual for details). A CS-directive is a <u>control directive</u>. A control directive affects an aspect of the processor's interpretation of those lines following the directive and preceding

either a subsequent directive modifying the same aspect, if such a directive appears, or the end of the program unit. The aspects of interpretation that may be controlled are:

(1) listing of the program and associated processor-produced information,

(2) specification of program lines to be processed or ignored,

(3) character data compare collation table, and

(4) minimum trip count for DO-loops.

Aspect (1) is _listing control_, aspect (2) is _conditional compilation_, aspect (3) is _collation control_, and aspect (4) is _DO-loop control_.

The general form of a control directive is

C$ _keyword_[(_p_[=_c_][,_p_[=_c_]]...)][[,]_lab_]

where:

keyword is one of LIST, IF, ELSE, ENDIF, COLLATE or DO. The keyword may begin in any column starting with column 7.

_p_ is a parameter. Depending upon the keyword that appears, one or more parameters may be specified.

_c_ is an integer constant, or symbolic name of an integer constant, with a value of zero or one. Depending upon the parameter _p_, the constant either may or must not appear.

lab is a _directive label_. Depending upon the keyword that appears, a directive label may be specified. If a directive label appears and no parameters are present, a comma must separate the keyword and the directive label.

The letter C in column 1 together with the character $ in column 2 identify a line as a C$-directive line. Such a line will be interpreted as a comment only if the appropriate FTN control statement option is selected. The entire directive must appear on a single line. A C$-directive interrupts statement continuation, unless it is being interpreted as a comment.

3.7.1 _LISTING CONTROL_

A listing control directive has the keyword LIST. It must have the

form:

```
    C$    LIST (p[=c][,p[=c]]...)
```

where:

p is S, O, R, A, M or ALL

c is a constant or the symbolic name of a constant

The constant is optional for all parameters; its absence is equivalent to the appearance of a constant with the value one. p=0 disables switch p; p=1 enables switch p.

ALL[=c] is equivalent to S[=c],O[=c],R[=c],A[=c],M[=c]

A listing control directive found by the processor to be in error results in a diagnostic message.

3.7.1.1 List Option Switches

The list option switches offer the following control

- S - source lines are listed when enabled.
- O - generated object code is listed for statements processed when enabled.
- R - symbol references are accumulated for the cross reference list when enabled. Symbols with no accumulated references will not appear in that list; no accumulation for an entire program unit suppresses cross reference list.
- A - the symbol attribute list is generated if this switch is enabled when the END statement is processed.
- M - the DO loop and common/equivalence map lists are generated if this switch is enabled when the END statement is processed.

3.7.2 CONDITIONAL COMPILATION

A conditional compilation directive has a keyword which is one of IF, ELSE or ENDIF. Such a directive controls whether the lines immediately following the directive are to be processed or ignored by the processor.

An _IF-directive_ has the form:

```
C$   IF(e)[[,]lab]
```

where:

- _e_ is an extended logical constant expression. If a symbolic name of a constant appears, it must have been previously defined in a PARAMETER statement of the program unit in which the IF-directive appears

- _lab_ is an optional directive label. It must be a symbolic name. Its use as a directive label in a conditional compilation directive does not affect, and is not affected by, its use for any other purpose in the program unit

An _ELSE_-directive has the form:

```
C$   ELSE[,lab]
```

where: _lab_ is as for an IF-directive

An _ENDIF-directive_ has the form:

```
C$   ENDIF[,lab]
```

where: _lab_ is as for an IF-directive

For each IF-directive there must appear exactly one ENDIF-directive later in the same program unit, and for each ENDIF-directive there must appear exactly one IF-directive earlier in the same program unit. Between an IF-directive and its corresponding ENDIF-directive will appear zero or more lines called a _conditional sequence_. A conditional sequence may optionally contain one ELSE-directive corresponding to the IF-directive and ENDIF-directive delimiting the conditional sequence. An ELSE-directive may appear only within a conditional sequence. A conditional sequence may not contain more than one ELSE-directive unless it contains another conditional sequence. If an ELSE-directive is contained within more than one conditional sequence, the ELSE-directive corresponds to that IF-ENDIF pair which delimits the smallest, i.e. innermost, conditional sequence containing the ELSE-directive.

If corresponding IF and ENDIF, or IF, ELSE, ENDIF directives have a directive label, it must be the same directive label. No other restriction applies to directive labels on conditional directives. There is no requirement that any conditional directive have a directive label. The same directive label may be used on more than one sequence of corresponding conditional directives in a single program unit, including the case of conditional directives whose conditional sequence contains other conditional directives with the

same directive label.

A conditional sequence may contain any number of properly corresponding conditional directives, and therefore other conditional sequences. If two conditional sequences contain the same line, one conditional sequence must lie wholly within the other conditional sequence.

A conditional compilation directive found by the processor to be in error results in a diagnostic message.

3.7.2.1 Processor Control

Processing of an IF-directive causes evaluation of the expression e.

If an IF-directive is processed by the processor and the value of e is true, following lines are processed as if the IF-directive had not appeared, unless a corresponding ELSE-directive is encountered. In this case, lines between the ELSE-directive and the corresponding ENDIF-directive are ignored by the processor. If an IF-directive is processed by the processor and the value of e is false, following lines are ignored until the corresponding ENDIF-directive as if they had not appeared, unless a corresponding ELSE-directive is encountered. In this case, lines between the ELSE-directive and the corresponding ENDIF-directive are processed. Note that conditional directives will be ignored if they appear within ignored sequences of lines. A line ignored by the processor must be a line acceptable to the processor if the line were not ignored.

3.7.3 COLLATION CONTROL

A collation control directive has the form

```
    C$   COLLATE(o)
```

where: o is FIXED or USER.

The collation control directive specifies whether collation of character relational expressions, and the values returned by the CHAR and ICHAR intrinsic functions, are directed by the fixed or user-specified weight table (3.1.5).

A collation control directive directs the interpretation of character relational expressions and of CHAR and ICHAR intrinsic

function references in the lines following the directive and preceding another collation control directive, if any, or the END statement of the program unit. In the case of a character relational expression or a CHAR or ICHAR intrinsic function reference in a statement function statement, the collation that applies is that in effect for the line or lines containing a reference to the statement function. Consider the example:

```
      PROGRAM P
      LOGICAL LSF
      CHARACTER*5, X, Y, S, T
C$    COLLATE(USER)
      LSF(X,Y) = X.LT.Y
              .
              .
              .
C$    COLLATE(FIXED)
      IF(LSF(S,T)) A=1.0
              .
              .
              .
      END
```

The reference LSF(S,T) results in an evaluation of the character relational expression S.LT.T with the collation that of the fixed weight table.

A collation control directive found by the processor to be in error results in a diagnostic message.

3.7.4 DO-LOOP CONTROL

A DO-loop control directive has the form

C$ DO(p[=c])

where:

p is OT

c is a constant or the symbolic name of a constant

The constant is optional; its absence is equivalent to the appearance of a constant with the value one.

The DO-loop control directive modifies the state of the DO-loop switch. The switch is initially set (or reset) when the

corresponding ONE_TRIP_DO option is (or is not) requested at processor invocation. A DO-loop control directive switch selection overrides the corresponding ONE_TRIP_DO option request.

The OT parameter controls the minimum trip count. If OT is set (OT=1), the minimum trip count for DO-loops is one. If OT is reset (OT=0), the minimum trip count for DO-loops is zero.

A DO-loop control directive affects the interpretation of only those DO-loops whose DO-statements follow the directive in the same program unit.

A DO-loop control directive found by the processor to be in error results in a diagnostic message.

4.0 DATA TYPES AND CONSTANTS

4.1 DATA TYPES

The nine types of data are:

(1) Integer

(2) Real

(3) Double precision

(4) Complex

(5) Logical

(6) Character

(7) Half precision

(8) Bit

(9) Boolean

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

4.1.1 DATA TYPE OF A NAME.

The name employed to identify a datum or a function also identifies its data type. A symbolic name representing a constant, variable, array, or function (except a generic function) must have only one type for each program unit. Once a particular name is identified with a particular type in a program unit, that type is implied for any usage of the name in the program unit that requires a type.

4.1.2 TYPE RULES FOR DATA AND PROCEDURE IDENTIFIERS.

A symbolic name that identifies a constant, variable, array, external function, or statement function may have its type specified in a type-statement (8.4) as integer, real, double precision, half precision, bit, complex, logical, Boolean, or character. In the absence of an explicit declaration in a type-statement, the type is implied by the first letter of the name. A first letter of I, J, K, L, M, or N implies type integer and any other letter implies type real, unless an IMPLICIT statement (8.5) is used to change the default implied type.

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

The data type of a function name specifies the type of the datum supplied by the function reference in an expression.

A symbolic name that identifies a specific intrinsic function in a program unit has a type as specified in 16.10. An explicit type-statement is not required; however, it is permitted. A generic function name does not have a predetermined type; the result of a generic function reference assumes a type that depends on the type of the argument, as specified in 16.10. If a generic function name appears in a type-statement, such an appearance is not sufficient by itself to remove the generic properties from that function.

In a program unit that contains an external function reference, the type of the function is determined in the same manner as for variables and arrays.

The type of an external function is specified implicitly by its name, explicitly in a FUNCTION statement, or explicitly in a type-statement. Note that an IMPLICIT statement within a function subprogram may affect the type of the external function specified in the subprogram.

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

4.1.3 DATA TYPE PROPERTIES.

The mathematical and representation properties for each of the data types are specified in the following sections. For real, double precision, half precision, and integer data, the value zero is considered neither positive nor negative. The value of a signed zero is the same as the value of an unsigned zero.

4.2 CONSTANTS

A *constant* is an arithmetic constant, logical constant, bit constant, Boolean constant, or character constant. The value of a constant does not change. Within an executable program, all constants that have the same form have the same value.

4.2.1 DATA TYPE OF A CONSTANT.

The form of the string representing a constant specifies both its value and data type. A PARAMETER statement (8.6) allows a constant to be given a symbolic name. The symbolic name of a constant must not be used to form part of another constant.

4.2.2 BLANKS_IN_CONSTANTS.

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

4.2.3 ARITHMETIC_CONSTANTS.

Integer, real, double precision, half precision, and complex constants are arithmetic constants.

4.2.3.1 Signs_of_Constants.

An unsigned constant is a constant without a leading sign. A signed constant is a constant with a leading plus or minus sign. An optionally signed constant is a constant that may be either signed or unsigned. Integer, real, double precision, and half precision constants may be optionally signed constants, except where specified otherwise.

4.3 INTEGER_TYPE

An integer datum is always an exact representation of an integer value. It may assume a positive, negative, or zero value. It may assume only an integral value. An integer datum has one numeric storage unit in a storage sequence.

4.3.1 INTEGER_CONSTANT.

The form of an integer constant is an optional sign followed by a nonempty string of digits. The digit string is interpreted as a decimal number.

4.4 REAL_TYPE

A real datum is a processor approximation to the value of a real number. It may assume a positive, negative, or zero value. A real datum has one numeric storage unit in a storage sequence.

4.4.1 BASIC_REAL_CONSTANT.

The form of a basic real constant is 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 of these parts may be omitted but not both. A basic real constant may be written with more digits than a processor will use to approximate the value of the constant. A basic real constant is interpreted as a decimal number.

4.4.2 REAL EXPONENT.

The form of a *real exponent* is the letter E followed by an optionally signed integer constant. A real exponent denotes a power of ten.

4.4.3 REAL CONSTANT.

The forms of a *real constant* are:

(1) Basic real constant

(2) Basic real constant followed by a real exponent

(3) Integer constant followed by a real exponent

The value of a real constant that contains a real exponent is the product of the constant that precedes the E and the power of ten indicated by the integer following the E. The integer constant part of form (3) may be written with more digits than a processor will use to approximate the value of the constant.

4.5 DOUBLE PRECISION TYPE

A double precision datum is a processor approximation to the value of a real number. The precision, although not specified, must be greater than that of type real. A double precision datum may assume a positive, negative, or zero value. A double precision datum has two consecutive numeric storage units in a storage sequence.

4.5.1 DOUBLE PRECISION EXPONENT.

The form of a *double precision exponent* is the letter D followed by an optionally signed integer constant. A double precision exponent denotes a power of ten. Note that the form and interpretation of a double precision exponent are identical to those of a real exponent, except that the letter D is used instead of the letter E.

4.5.2 DOUBLE PRECISION CONSTANT.

The forms of a *double precision constant* are:

(1) Basic real constant followed by a double precision exponent

(2) Integer constant followed by a double precision exponent

The value of a double precision constant is the product of the constant that precedes the D and the power of ten indicated by the integer following the D. The integer constant part of form (2) may be written with more digits than a processor will use to approximate the value of the constant.

4.6 COMPLEX TYPE

A complex datum is a processor approximation to the value of a complex number. The representation of a complex datum is in the form of an ordered pair of real data. The first of the pair represents the real part of the complex datum and the second represents the imaginary part. Each part has the same degree of approximation as for a real datum. A complex datum has two consecutive numeric storage units in a storage sequence; the first storage unit is the real part and the second storage unit is the imaginary part.

4.6.1 COMPLEX CONSTANT

The form of a complex constant is a left parenthesis followed by an ordered pair of real or integer constants, or symbolic names of real or integer constants, separated by a comma, and followed by a right parenthesis. The first constant, or symbolic name of a constant, of the pair is the real part of the complex constant and the second is the imaginary part.

4.7 LOGICAL TYPE

A logical datum may assume only the values true or false. A logical datum has one numeric storage unit in a storage sequence.

4.7.1 LOGICAL CONSTANT.

The forms and values of a logical constant are:

Form	Value
.TRUE.	true
.FALSE.	false

4.8 CHARACTER TYPE

A character datum is a string of characters. The 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. A character datum has one character storage unit in a storage sequence for each character in the string. The maximum number of characters in a character datum is 2**16-1 (2**16-1 = 65,535).

Each character in the string has a character position that is numbered consecutively 1, 2, 3, etc. The number indicates the sequential position of a character in the string, beginning at the

left and proceeding to the right.

4.8.1 CHARACTER CONSTANT.

The form of a _character constant_ is an apostrophe followed by a nonempty string of characters followed by an apostrophe. The string may consist of any characters capable of representation in the processor. Note that the delimiting apostrophes are not part of the datum represented by the constant. An apostrophe within the datum string is represented by two consecutive apostrophes with no intervening blanks. In a character constant, blanks embedded between the delimiting apostrophes are significant.

The length of a character constant is the number of characters 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.

4.9 BOOLEAN TYPE

A Boolean datum is a string of _n_ bits, where _n_ is the number of bit positions belonging to one numeric storage unit. A Boolean datum has one numeric storage unit.

4.9.1 BOOLEAN CONSTANT

A _Boolean constant_ is a Hollerith constant, an octal constant, or a hexadecimal constant.

4.9.1.1 Hollerith Constant

A _Hollerith constant_ has one of four forms:

_n_H_f_
L"_f_"
R"_f_"
"_f_"

where:

_n_ is an unsigned, nonzero, integer constant not greater than _g_, where _g_ is the maximum number of characters that can be represented by the processor in one numeric

storage unit.

f is a string of n characters. The string may consist of any characters capable of representation in the processor. Blanks in a Hollerith constant are significant only in this string.

The datum represented by a Hollerith constant is determined by the internal processor code for the characters in the string f and the form of fill indicated by the form of the constant. By fill is meant the string of bit values supplied by the processor either to the left or right of f, if any is required, to complete the datum. The datum is complete when all its bits are determined.

For the nHf and "f" forms, the string f is left-justified in the datum and the fill is a string of bit values constituting the internal processor code for a string of blank characters.

For the L"f" form, the string f is left-justified in the datum and the fill is a string of 0 bit values.

For the R"f" form, the string f is right-justified in the datum and the fill is a string of 0 bit values.

The character " within "f" is represented by a string of two consecutive " characters within f.

4.9.1.2 Octal Constant

An octal constant has the form:

O"b[b]..."

where: b is an octal digit.

The datum represented by an octal constant is the string of bit values obtained by mapping each octal digit into its equivalent three-digit binary number system value, right-justified with a fill which is a string of 0 bit values. The number of bit values in the resulting string must not be greater than the number of bit positions in one numeric storage unit. Note that a blank character is not significant anywhere within an octal constant.

4.9.1.3 Hexadecimal_Constant

A hexadecimal_constant has the form:

Z"z[z]..."

where: z is a hexadecimal digit.

The datum represented by a hexadecimal constant is the string of bit values obtained by mapping each hexadecimal digit into its equivalent four-digit binary number system value, right-justified with a fill which is a string of 0 bit values. The number of bit values in the resulting string must not be greater than the number of bit positions in one numeric storage unit. Note that a blank character is not significant anywhere within a hexadecimal constant.

4.10 HALF_PRECISION_TYPE

A half precision datum is a processor approximation to the value of a real number. A half precision datum may assume a positive, negative, or zero value. A half precision datum has one-half of a numeric storage unit in a storage sequence.

4.10.1 HALF_PRECISION_EXPONENT

The form of a half precision exponent is the letter S followed by an optionally signed integer constant. A half precision exponent denotes a power of ten. Note that the form and interpretation of a half precision exponent are identical to those of a real exponent, or double precision exponent, except that the letter S is used instead of the letter E, or D.

4.10.2 HALF_PRECISION_CONSTANT

The forms of a half precision constant are:

(1) basic real constant followed by a half precision exponent

(2) integer constant followed by a half precision exponent

The value of a half precision constant is the product of the constant that precedes the S and the power of ten indicated by the integer following the S. The integer part of form (2) may be

written with more digits than a processor will use to approximate the value of the constant.

4.11 BIT TYPE

A bit datum may assume only the values B"0" and B"1". A bit datum has one bit storage unit in a storage sequence.

4.11.1 BIT CONSTANT

The forms of a bit constant are B"0" and B"1". Note that a blank character is not significant anywhere within a bit constant.

5.0 ARRAYS AND SUBSTRINGS

An _array_ is a (possibly empty) sequence of data. An _array element_ is one member of the sequence of data. An _array name_ is the symbolic name of an array. An _array element name_ is an array name qualified by a subscript (5.4).

An array name not qualified by a subscript identifies the entire sequence of elements of the array in certain forms where such use is permitted (5.9); however, in an EQUIVALENCE statement, an array name not qualified by a subscript identifies the first element of the array (8.2.4).

An array element name identifies one element of the sequence. The subscript value (Table 1) specifies the element of the array being identified. A different array element may be identified by changing the subscript value of the array element name.

An array name is local to a program unit (19.1.2).

A _substring_ is a contiguous portion of a character datum.

5.1 ARRAYS AND ARRAY SECTIONS

5.1.1 ARRAY NAME AND ARRAY SECTION REFERENCE

Whenever an array name, unqualified by a subscript, or an array section name is used to designate the whole array, or an array section, the appearance of the array name or array section name implies that the number of values to be processed is equal to the number of elements in the array or array section and that the elements of the array or array section are to be processed in sequential order, except in the evaluation of array expressions and in array assignment statements.

An assumed size array may not be used in an array reference. An assumed size array section reference may not be used in an array reference if a section selector of * or -* is used in the last subscript position.

5.2 ARRAY DECLARATOR

An _array declarator_ specifies a symbolic name that identifies an array within a program unit and specifies certain properties of the array. Only one array declarator for an array name is permitted in a program unit.

5.2.1 FORM OF AN ARRAY DECLARATOR

The form of an array declarator is:

a (d [,d]...)

where: a 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.

5.2.1.1 Form of a Dimension Declarator

The form of a dimension declarator is one of the forms:

[d1:] d2
[d1]:

where: d1 is the lower dimension bound

d2 is the upper dimension bound

The lower and upper dimension bounds are arithmetic or Boolean scalar expressions, called dimension bound expressions, in which each primary is a constant, symbolic name of constants, dummy argument name of a variable or array, or any intrinsic function reference which is allowed in an extended integer constant expression. A dimension bound expression must be scalar and of integer type. An actual argument to an intrinsic function reference in a dimension bound expression may be any entity that is allowed in an extended constant expression or that is allowed in a dimension bound expression for an adjustable array declarator. The upper dimension bound of the last dimension must be an asterisk in an assumed-size array declarator. A dimension bound expression must not contain a nonintrinsic function or array element reference. Variables may appear in dimension bound expressions only in adjustable array declarators and as actual arguments to intrinsic function references. Array names may appear in dimension bound expressions only as actual arguments to intrinsic function references.

In a dimension bound expression, the symbolic name of a constant, a variable, or an array must be explicitly typed prior to its appearance in the dimension bound expression. If it is not explicitly typed prior to its appearance in the dimension bouncd expression, it will be typed using its initial letter and the default implied type rules then in effect. If an entity in a dimension bound expression is typed by implied typing, any

subsequent explicit typing must confirm the implied type.

No array name appearing in an array declarator in a specification statement may be referenced in a primary of any expression within the same specification statement. For example,

```
      REAL A(10,EXTENT(A,1))
```

is prohibited.

5.2.1.2 Value of Dimension Bounds

If a dimension bound di is a Boolean expression, the value used is INT(di). The value of either dimension bound may be positive, negative, or zero. If only the upper dimension bound is specified, the value of the lower dimension bound is one. If the dimension declarator is of the form [d1]: and the lower dimension bound is omitted, its value is one. An upper dimension bound of an asterisk is always greater than or equal to the lower dimension bound.

5.2.2 KINDS AND OCCURRENCES OF ARRAY DECLARATORS

Each array declarator is either a constant array declarator, an adjustable array declarator, an assumed size array declarator, or an assumed-shape array declarator. A constant array declarator is an array declarator in which each of the dimension bound expressions is an integer constant expression (6.1.3.1). An adjustable array declarator is an array declarator that contains one or more variables. An assumed-size array declarator is a columnwise assumed-size array declarator or a rowwise assumed-size array declarator. A columnwise assumed-size array declarator is a constant array declarator or an adjustable array declarator, except that the upper dimension bound of the last dimension is an asterisk. A rowwise assumed-size array declarator is a constant array declarator or an adjustable array declarator, except that the upper dimension bound of the first dimension is an asterisk. An assumed-shape array declarator is a constant array declarator or an adjustable array declarator, except that one or more of the dimension declarators has a specificaton of the upper bound omitted.

Each array declarator is either an actual array declarator, a dummy array declarator, or an allocatable array declarator.

5.2.2.1 Actual Array Declarator

An actual array declarator is an array declarator in which the array name is not a dummy argument. Each actual array declarator must be a constant array declarator. An actual array declarator is permitted in a DIMENSION statement, a type-statement, or a COMMON statement (Section 8).

5.2.2.2 Dummy Array Declarator

A dummy array declarator is an array declarator in which the array name is a dummy argument or is the name of the array-valued function in which the array is being declared. A dummy array declarator may be a constant array declarator, an adjustable array declarator, an assumed-shape array declarator, or an assumed-size array declarator. A dummy array declarator is permitted in a DIMENSION statement or a type-statement but not in a COMMON statement. A dummy array declarator may appear only in a function or subroutine subprogram.

5.2.2.3 Allocatable Array Declarator

An allocatable array declarator is an array declarator in which the array name is not a dummy argument. An allocatable array declarator must be an assumed-shape array declarator with both upper and lower dimension bounds omitted for every dimension.

5.3 PROPERTIES OF AN ARRAY

The following properties of an array are specified by the array declarator: the number of dimensions of the array, the size and bounds of each dimension, and therefore the number of array elements.

The properties of an array in a program unit are specified by the array declarator for the array in that program unit.

5.3.1 DATA TYPE OF AN ARRAY AND AN ARRAY ELEMENT

An array name has a data type (4.1.1). An array element name has the same data type as the array name.

5.3.2 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:

MAX(d2-d1+1, 0)

where: d1 is the value of the lower dimension bound

d2 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 MAX(d2, 0).

The size of a dimension whose upper bound is an asterisk is not specified. The size of a dimension of an assumed-shape array for

which the dimension bound expression is of the form [*d1*]: is MAX(*sa*-*d1*-1,0) where *sa* is the size of the corresponding dimension of the associated actual argument.

The number and size of dimensions in one array declarator may be different from the number and size of dimensions in another array declarator that is associated by common, equivalence, or argument association, except as prohibited by the table in section 8.2.3 and by section 16.9.3.3.

5.3.3 SIZE OF AN ARRAY

The *size of an array* is equal to the number of elements in the array. The size of an array is equal to the product of the sizes of the dimensions specified by the array declarator for that array name. The size of an assumed-size dummy array (5.8) is determined as follows:

(1) If the actual argument corresponding to the dummy array is a noncharacter array name, the size of the dummy array is the size of the actual argument array.

(2) If the actual argument corresponding to the dummy array name is a noncharacter array element name with a subscript value of r in an array of size x, the size of the dummy array is $x + 1 - r$.

(3) If the actual argument is a character array name, character array element name, or character array element substring name and begins at character storage unit t of an array with c character storage units, then the size of the dummy array is INT((c + 1 - t) / ln), where ln is the length of an element of the dummy array.

If a columnwise assumed-size dummy array has n dimensions, the product of the sizes of the first n-1 dimensions must be less than or equal to the size of the array, as determined by one of the immediately preceding rules. If a rowwise assumed-size dummy array has n dimensions, the product of the sizes of the last n-1 dimensions must be less than or equal to the size of the array, as determined by one of the immediately preceding rules.

The size of an allocatable array is specified by an ALLOCATE statement.

5.3.4 ARRAY ELEMENT ORDERING

The elements of an array are ordered in a sequence (2.1). An array element name contains a subscript (5.5.1) whose subscript value (5.5.3) determines which element of the array is identified by the

array element name. 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 an array name unqualified by a subscript is used to designate the whole array (5.9), 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 to be taken in sequential order except in the evaluation of array expressions, in array assignment statements, and in FORALL statements.

5.3.5 ARRAY_STORAGE_SEQUENCE

An array has a storage sequence consisting of the storage sequences of the array elements in the order determined by the array element ordering. The number of storage units in an array is $x*z$, where x is the number of the elements in the array and z is the number of storage units for each array element.

5.4 ARRAY_ELEMENT_NAME

The form of an array element name is:

a (s [,s]...)

where: a is the array name

(s [,s]...) is a subscript (5.5.1)

s is a subscript expression (5.5.2)

The number of subscript expressions must be equal to the number of dimensions in the array declarator for the array name.

5.5 SUBSCRIPT

5.5.1 FORM_OF_A_SUBSCRIPT

The form of a subscript is:

(s [,s]...)

where s is a subscript expression.

Note that the term "subscript" includes the parentheses that delimit the list of subscript expressions.

5.5.2 SUBSCRIPT EXPRESSION

A subscript expression is an integer, real, double precision, half precision, complex, or Boolean scalar expression. A subscript expression may contain array element references and function references. Note that a restriction in the evaluation of expressions (6.6) prohibits certain side effects. In particular, evaluation of a function must not alter the value of any other subscript expression within the same subscript.

Within a program unit, the value of each subscript expression must be greater than or equal to the corresponding lower dimension bound in the array declarator for the array. The value of each subscript expression must not exceed the corresponding upper dimension bound declared for the array in the program unit. If the upper dimension bound is an asterisk, the value of the corresponding subscript expression must be such that the subscript value does not exceed the size of the dummy array. If an upper dimension bound is omitted in an assumed-shape array, the value of the corresponding subscript expression si must be such that

$$si \le dl + sa - 1$$

where dl is the lower dimension bound of the dimension where the subscript is written and sa is the size of the corresponding dimension of the actual argument associated with the assumed-shape array.

The value of each subscript expression of an allocatable array must be greater than or equal to the corresponding lower dimension bound established by the execution of the ALLOCATE statement. The value of each subscript expression of an allocatable array must not exceed the corresponding upper dimension bound established by the execution of the ALLOCATE statement.

5.5.3 SUBSCRIPT VALUE

The subscript value of a subscript is specified in Table 1. The subscript value determines which array element is identified by the array element name. Within a program unit, the subscript value depends on the values of the subscript expressions in the subscript, the dimensions of the array specified in the array declarator for the array in the program unit, and whether the array is a columnwise array or a rowwise array (see 8.10). If the subscript value is r, the rth element of the array is identified.

Table 1
Subscript Value
For Columnwise Array

n	Dimension Declarator	Subscript	Subscript Value
1	(j1:k1)	(s1)	1+(s1-j1)
2	(j1:k1,j2:k2)	(s1,s2)	1+(s1-j1) +(s2-j2)*d1
3	(j1:k1,j2:k2,j3:k3)	(s1,s2,s3)	1+(s1-j1) +(s2-j2)*d1 +(s3-j3)*d2*d1
. . .	. . .	. . .	. . .
n	(j1:k1,...,jn:kn)	(s1,...,sn)	1+(s1-j1) +(s2-j2)*d1 +(s3-j3)*d2*d1 +... +(sn-jn)*dn-1 *dn-2*...*d1

Table 1 (continued)
Subscript Value
For Rowwise Array

n	Dimension Declarator	Subscript	Subscript Value
1	(j1:k1)	(s1)	1+(s1-j1)
2	(j1:k1, j2:k2)	(s1,s2)	1+(s1-j1)*d2 +(s2-j2)
3	(j1:k1, j2:k2, j3:k3)	(s1,s2,s3)	1+(s1-j1)*d2*d3 +(s2-j2)*d3 +(s3-j3)
. . .	. . .	. . .	. . .
n	(j1:k1,...,jn:kn)	(s1,...,sn)	1+(s1-j1)*d2 *d3*...*dn +(s2-j2)*d3 *d4*...*dn +... +(sn-1-jn-1)*dn +(sn-jn)

Notes for Table 1:

(1) n is the number of dimensions, $1 \le n \le 7$

(2) ji is the value of the lower bound of the ith dimension

(3) ki is the value of the upper bound of the ith dimension

(4) If only the upper bound is specified, then ji = 1

(5) si is the integer value of the ith subscript expression

(6) di = ki-ji+1 is the size of the ith dimension. If the value of the lower bound is 1, then di = ki

Note that for arrays that do not have size zero a subscript of the form (j1,...,jn) has a subscript value of one and identifies the first element of the array. A subscript of the form (k1,...,kn) identifies the last element of the array; its subscript value is equal to the number of elements in the array.

The subscript value and the subscript expression value are not necessarily the same, even for a one-dimensional array. In the example:

```
DIMENSION A(-1:8),B(10,10)
A(2) = B(1,2)
```

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

5.6 ARRAY SECTION NAMES

The form of an array section name is:

a(ss[,ss]...)

where:

a	is the array name
(ss[,ss]...)	is a section subscript
ss	is a section subscript designator

The number of section subscript designators must be equal to the number of dimensions in the array declarator for the array name.

5.6.1 SECTION SUBSCRIPT

5.6.1.1 Form of a Section Subscript

The form of a *section subscript* is:

(ss[,ss]...)

where ss is a section subscript designator. At least one of the ss must be a section selector.

Note that the term *section subscript* includes the parentheses that

delimit the list of subscript expressions.

5.6.1.2 Section Subscript Expression

A section subscript expression is a subscript expression or a section selector. A section selector is an indexed section selector or a vector-valued section selector. An indexed section selector is of the form:

```
[ss1]:[ss2][:ss3]
```

where ss is a subscript expression. A vector-valued section selector is a one-dimensional integer array expression.

If ss1 is omitted, then the value of the corresponding lower dimension bound is implied for ss1. If ss2 is omitted, then the value of the corresponding upper dimension bound is implied for ss2. If ss3 is omitted, then the value of one is implied for ss3.

5.7 ARRAY SECTION

An array section is a subsequence of an array. The number of dimensions of the array section is equal to the number of section selectors.

If an array is rowwise (columnwise), then any section thereof is also considered to be rowwise (columnwise).

Each section selector identifies elements for the dimension position where it is written. The order of dimensions in an array section is determined from left to right by the appearance of section selectors. For example,

```
A(*,2,*)
```

is an array section with two dimensions. The first dimension corresponds to the first dimension of A and the second dimension corresponds to the third dimension of A.

The total number of elements identified is given by the value of the expression

```
MAX(INT((ss2-ss1+ss3)/ss3),0)
```

If the total number of elements is positive then the section selector identifies the elements from ss1 to ss2 in increments of ss3 and each element must lie within the bounds of the array being sectioned. The size of a dimension of an array section is equal to the number of elements identified in that dimension. The size of an array section is equal to the product of the sizes of the dimensions of the array section. In usages where an ordering of elements of an

array section is implied, for example, in an output list, it is as if the array section is formed and a new temporary array created and the array elements of the temporary array are used in order of subscript value (5.5.3).

A vector-valued section selector identifies a section formed by selecting the elements identified by the values of the elements of the one dimensional array used as a section selector. The values of elements of the one dimensional array used as a section selector must not select elements outside the bounds of the array being sectioned.

When a vector-valued section is used, the number of elements in the resulting section along the dimension selected by the vector-valued expression is determined by and is equal to the number of elements in the vector-valued expression.

An array reference involving vector-valued subscripts, when used as an actual argument is an expression and may not be defined within the subprogram.

For example, suppose Z is a two dimensional array of 5 by 7 elements and U and V are one dimensional arrays of 3 and 4 elements, respectively. Assume the values of U and V are:

```
U = 1 3 2
V = 2 1 1 3
```

then Z(3,V) consists of the elements of the third row of Z in the order:

```
Z(3,2)Z(3,1)Z(3,1)Z(3,3)
```

and Z(U,2) consists of the column elements:

```
Z(1,2)Z(3,2)Z(2,2)
```

and finally Z(U,V) consists of the elements:

```
Z(1,2)Z(1,1)Z(1,1)Z(1,3)
Z(3,2)Z(3,1)Z(3,1)Z(3,3)
Z(2,2)Z(2,1)Z(2,1)Z(2,3)
```

5.8 DUMMY AND ACTUAL ARRAYS

A *dummy array* is an array for which the array declarator is a dummy array declarator. An *assumed-size dummy array* is a dummy array for which the array declarator is an assumed-size array declarator. An *assumed-shape* dummy array is a dummy array for which the array declarator is an assumed-shape declarator. A dummy array is permitted only in a function or subroutine subprogram (Section 16).

An _actual array_ is an array for which the array declarator is an actual array declarator or an allocatable array declarator. Each array in the main program must be an actual array. An actual array in a subprogram must have a constant array declarator or an allocatable array declarator. An actual array with an allocatable array declarator is called an _allocatable array_. A dummy array may be used as an actual argument.

5.8.1 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 contain a variable name are called _adjustable dimensions_.

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 bound expression of an array must also appear as a name either in every dummy argument list that contains the array name or in a common block in that subprogram.

At the time of execution of a reference to a function or subroutine containing an adjustable array in its dummy argument list, each actual argument that corresponds to a dummy argument appearing in a dimension bound expression for the array and each variable in common appearing in a dimension bound expression for the array must be defined with an integer value. The values of those dummy arguments or variables in common, together with any constants and symbolic names of constants appearing in the dimension bound expression, determine the size of the corresponding adjustable dimension for the execution of the subprogram. The sizes of the adjustable dimensions and of any constant dimensions appearing in an adjustable array declarator determine the number of elements in the 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 the subprogram. These properties depend on the values of any actual arguments and variables in common that are referenced in the adjustable dimension expressions in the subprogram.

During the execution of an external procedure in a subprogram containing an adjustable array, the array properties of dimension size, lower and upper dimension bounds, and array size (number of elements in the array) do not change. However, the variables involved in an adjustable dimension may be redefined or become undefined during execution of the external procedure with no effect on the above mentioned properties.

5.9 USE OF ARRAY NAMES

In a program unit, each appearance of an array name must be in an array element name except in the following cases:

(1) In a list of dummy arguments

(2) In a COMMON statement

(3) In a type-statement

(4) In an array declarator. Note that although the form of an array declarator may be identical to that of an array element name, an array declarator is not an array element name.

(5) In an EQUIVALENCE statement

(6) In a DATA statement

(7) In the list of actual arguments in a reference to an external procedure or an intrinsic function

(8) In the list of an input/output statement

(9) As a unit identifier for an internal file in an input/output statement

(10) As the format identifier in an input/output statement

(11) In a SAVE statement

(12) As a primary in an array expression

(13) In an array assignment statement

(14) In an ALLOCATE or FREE statement

(15) In an IDENTIFY statement

(16) In a FORALL statement

5.9.1 APPEARANCE OF ARRAY SECTION NAMES

In a program unit, array section names may appear in the following places:

(1) In the list of actual arguments in a reference to an external procedure or an intrinsic function

(2) In the list of an input/output statement

(3) As a unit identifier for an internal file in an input/output statement

(4) As the format identifier in an input/output statement

(5) As a primary in an array expression

(6) In an array assignment statement

(7) In a FORALL statement

5.10 CHARACTER SUBSTRING

A character substring is a contiguous portion of a character datum and is of type character. A character substring is identified by a substring name and may be assigned values and referenced.

5.10.1 SUBSTRING NAME.

The forms of a *substring name* are:

v ([*e1*] : [*e2*])

a (*s* [,*s*]...)([*e1*] : [*e2*])

where: *v* is a character variable name

a (*s* [,*s*]...) is a character array element name or array section name

e1 and *e2* are each an integer, real, double precision, half precision, complex, or Boolean scalar expression and are called *substring expressions*. If *a*(*s*[,*s*]...) is an array section, it identifies the substrings of the array elements identified by the section subscript.

The value *e1* specifies the leftmost character position of the substring and the value *e2* specifies the rightmost character position. For example, A(2:4) specifies characters in positions two through four of the character variable A, and B(4,3)(1:6) specifies characters in positions one through six of the character array element B(4,3).

The length of a character substring is MAX(0,*e2*-*e1*+1). If the length is non-zero then the values of *e1* and *e2* must be such that $1 \le e1 \le len$, $1 \le e2 \le len$. If *e1* is omitted, a value of one is implied for *e1*. If *e2* is omitted, a value of *len* is implied for *e2*. Both *e1* and *e2* may be omitted; for example, the form *v*(:) is equivalent to *v*, and the form *a*(*s* [,*s*]...)(:) is equivalent to *a*(*s* [,*s*]...).

5.10.2 SUBSTRING EXPRESSION.

A *substring expression* may be any scalar integer, real, double precision, half precision, complex or Boolean expression. A substring expression may contain array element references and function references. Note that a restriction in the evaluation of expressions (6.6) prohibits certain side effects. In particular, evaluation of a function must not alter the value of any other expression within the same substring name.

6.0 EXPRESSIONS

This section describes the formation, interpretation, and evaluation rules for arithmetic, character, relational, Boolean, bit, and logical expressions. An expression is formed from operands, operators, and parentheses.

An _array operand_ is an array reference, array section reference, or a function reference whose value is an array. An _array expression_ is an arithmetic expression, relational expression, character expression, logical expression, Boolean expression, or bit expression in which one or more primaries is an array operand. An array expression produces an array-valued result.

6.1 ARITHMETIC EXPRESSIONS

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

The simplest form of an arithmetic expression is an unsigned arithmetic constant, symbolic name of an arithmetic constant, arithmetic variable reference, arithmetic array element reference, arithmetic array reference, arithmetic array section reference, or arithmetic function reference. More complicated arithmetic expressions may be formed by using one or more arithmetic or Boolean operands together with arithmetic operators and parentheses. Arithmetic operands must identify values of type integer, real, double precision, half precision, or complex.

6.1.1 ARITHMETIC OPERATORS.

The five arithmetic operators are:

Operator	Representing
**	Exponentiation
/	Division
*	Multiplication
-	Subtraction or Negation
+	Addition or Identity

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.

6.1.2 FORM AND INTERPRETATION OF ARITHMETIC EXPRESSIONS

The interpretation of the expression formed with each of the arithmetic operators in each form of use is as follows:

Use of Operator	Interpretation
x1 ** x2	Exponentiate x1 to the power x2
x1 / x2	Divide x1 by x2
x1 * x2	Multiply x1 and x2
x1 - x2	Subtract x2 from x1
- x2	Negate x2
x1 + x2	Add x1 and x2
+ x2	Same as x2

where: x1 denotes the operand to the left of the operator

x2 denotes the operand to the right of the operator

The interpretation of a division may depend on the data types of the operands (6.1.5).

A set of formation rules is used to establish the interpretation of an arithmetic expression that contains two or more operators. There is a precedence among the arithmetic operators, which determines the order in which the operands are to be combined unless the order is changed by the use of parentheses. The precedence of the arithmetic operators is as follows:

Operator	Precedence
**	Highest
* and /	Intermediate
+ and -	Lowest

For example, in the expression

```
- A ** 2
```

the exponentiation operator (**) has precedence over the negation operator (-); therefore, the operands of the exponentiation operator are combined to form an expression that is used as the operand of the negation operator. The interpretation of the above expression is the same as the interpretation of the expression

- (A ** 2)

The _arithmetic operands_ are:

(1) Primary

(2) Factor

(3) Term

(4) Arithmetic expression

The formation rules to be applied in establishing the interpretation of arithmetic expressions are in 6.1.2.1 through 6.1.2.4.

6.1.2.1 _Primaries._

The _primaries_ are:

(1) Unsigned arithmetic constant (4.2.3)

(2) Symbolic name of an arithmetic constant (8.6)

(3) Arithmetic variable reference (2.5)

(4) Arithmetic array element reference (5.4)

(5) Arithmetic array reference

(6) Arithmetic array section reference

(7) Arithmetic function reference (16.2)

(8) Arithmetic expression enclosed in parentheses (6.1.2.4)

6.1.2.2 _Factor._

The forms of a _factor_ are:

(1) Primary

(2) Primary ** factor

(3) Boolean primary ** factor

(4) Primary ** Boolean primary

(5) Boolean primary ** Boolean primary

Thus, a factor is formed from a sequence of one or more primaries or Boolean primaries (6.7.1.1) separated by the exponentiation operator. Form (2) indicates that in interpreting a factor containing two or more exponentiation operators, the primaries are combined from right to left. For example, the factor

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

has the same interpretation as the factor

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

6.1.2.3 Term.

The forms of a term are:

(1) Factor

(2) Term / factor

(3) Term * factor

(4) Term / Boolean primary

(5) Term * Boolean primary

(6) Boolean primary / factor

(7) Boolean primary * factor

(8) Boolean primary / Boolean primary

(9) Boolean primary * Boolean primary

Thus, a term is formed from a sequence of one or more factors or Boolean primaries (6.7.1.1) separated by either the multiplication operator or the division operator. The above forms indicate that in interpreting a term containing two or more multiplication or division operators, the factors or Boolean primaries are combined from left to right.

6.1.2.4 Arithmetic Expression.

The forms of an arithmetic expression are:

(1) Term

(2) + term

(3) - term

(4) Arithmetic expression + term

(5) Arithmetic expression - term

(6) + Boolean primary

(7) - Boolean primary

(8) Arithmetic expression + Boolean primary

(9) Arithmetic expression - Boolean primary

(10) Boolean primary + term

(11) Boolean primary - term

(12) Boolean primary + Boolean primary

(13) Boolean primary - Boolean primary

Thus, an arithmetic expression is formed from a sequence of one or more terms or Boolean primaries separated by either the addition operator or the subtraction operator. The first term or Boolean primary in an arithmetic expression may be preceded by the identity or the negation operator. The above forms indicate that in interpreting an arithmetic expression containing two or more addition or subtraction operators, the terms or Boolean primaries are combined from left to right.

Note that these formation rules do not permit expressions containing two consecutive arithmetic operators, such as A**-B or A+-B. However, expressions such as A**(-B) and A+(-B) are permitted.

6.1.3 ARITHMETIC CONSTANT EXPRESSION.

An *arithmetic constant expression* is an arithmetic expression in which each (arithmetic) primary is an arithmetic constant, the symbolic name of an arithmetic constant, or an arithmetic constant expression enclosed in parantheses, and each Boolean primary (6.7.1.1) is a Boolean constant, the symbolic name of a Boolean constant, or a Boolean constant expression enclosed in parentheses. The exponentiation operator is not permitted unless the exponent is of type integer or Boolean. If the exponent *e* is of type Boolean, the value used is INT(*e*). Note that variable, array element, and function references are not allowed.

An *extended* arithmetic constant expression is an arithmetic constant

expression except:

(1) selected elemental intrinsic functions are allowed, when referenced with constant arguments,

(2) selected array-valued intrinsic functions are allowed.

The list of allowed elemental and array-valued intrinsic functions is implementation dependant.

6.1.3.1 Integer Constant Expression.

An *integer constant expression* is an arithmetic constant expression (6.1.3) or a Boolean constant expression (6.7.3) in which each constant or symbolic name of a constant is of type integer or Boolean. If the integer constant expression *e* is a Boolean constant expression, the value used is INT(*e*). Note that variable, array element, and function references are not allowed.

The following are examples of integer constant expressions:

```
 3
-3
-3+4
O"74"
R"A"
R"AB" .AND. 48
```

An *extended* integer constant expression is an integer constant expression except:

(1) selected elemental intrinsic functions are allowed, when referenced with constant arguments,

(2) selected array-valued intrinsic functions are allowed.

The list of allowed elemental and array-valued intrinsic functions is implementation dependant.

6.1.4 TYPE AND INTERPRETATION OF ARITHMETIC EXPRESSIONS

The data type of a constant is determined by the form of the constant (4.2.1). The data type of an arithmetic variable reference, symbolic name of an arithmetic constant, arithmetic array element reference, or arithmetic function reference is determined by the name of the datum or function (4.1.2). The data type of an arithmetic expression containing one or more arithmetic operators is determined from the data types of the operands.

Integer expressions, *real expressions*, *double precision expressions*, *half precision expressions*, and *complex expressions* are arithmetic

expressions whose values are of type integer, real, double precision, half precision, and complex, respectively.

When the operator + or - operates on a single operand, the data type of the resulting expression is the same as the data type of the operand unless the operand is of type Boolean, in which case the type of the resulting expression is integer.

When an arithmetic operator operates on a pair of operands, the data type of the resulting expression is given in Tables 2 and 3. In these tables, each letter I, R, D, H, or C represents an operand or result of type integer, real, double precision, or complex, respectively.

The type of the result is indicated by the I, R, D, H, or C that precedes the equals, and the interpretation is indicated by the expression to the right of the equals. REAL, DBLE, HALF, EXTEND, and CMPLX are the type-conversion functions described in 16.10.

Table 2

Type and Interpretation of Result for x1+x2

x2 / x1	I2	R2
I1	I = I1 + I2	R = REAL(I1) + R2
R1	R = R1 + REAL(I2)	R = R1 + R2
D1	D = D1 + DBLE(I2)	D = D1 + DBLE(R2)
H1	H = H1 + HALF(I2)	R = REAL(H1) + R2
C1	C=C1+CMPLX(REAL(I2),0.)	C = C1 + CMPLX(R2,0.)

x2 / x1	D2	C2
I1	D = DBLE(I1) + D2	C=CMPLX(REAL(I1),0.)+C2
R1	D = DBLE(R1) + D2	C = CMPLX(R1,0.) + C2
D1	D = D1 + D2	C=CMPLX(REAL(D1),0.)+C2
H1	D = DBLE(H1) + D2	C=CMPLX(REAL(H1),0.)+C2
C1	C=C1+CMPLX(REAL(D2),0.)	C = C1 + C2

x2 / x1	H2
I1	H = HALF(I1) + H2
R1	R = R1 + REAL(H2)
D1	D = D1 + DBLE(H2)
H1	H = H1 + H2
C1	C=C1+CMPLX(REAL(H2),0.)

Tables giving the type and interpretation of expressions involving -, *, and / may be obtained by replacing all occurrences of + in Table 2 by -, *, or /, respectively.

Table 3

Type and Interpretation of Result for x1**x2

x2 / x1	I2	R2
I1	I = I1**I2	R = REAL(I1)**R2
R1	R = R1**I2	R = R1**R2
D1	D = D1**I2	D = D1**DBLE(R2)
H1	H = H1**I2	R = REAL(H1)**R2
C1	C = C1**I2	C = C1**CMPLX(R2,0.)

x2 / x1	D2	C2
I1	D = DBLE(I1)**D2	C=CMPLX(REAL(I1),0.)**C2
R1	D = DBLE(R1)**D2	C = CMPLX(R1,0.)**C2
D1	D = D1** D2	C=CMPLX(REAL(D1),0.)**C2
H1	D = DBLE(H1)**D2	C=CMPLX(REAL(H1),0.)**C2
C1	C=C1**CMPLX(REAL(D2),0.)	C = C1**C2

x2 / x1	H2
I1	H = HALF(I1)**H2
R1	R = R1**REAL(H2)
D1	D = D1**DBLE(H2)
H1	H = H1**H2
C1	C=C1**CMPLX(REAL(H2),0.)

Five entries in Table 3 specify an interpretation to be a complex value raised to a complex power. In these cases, the value of the

expression is the "principal value" determined by x1**x2 = EXP(x2*LOG(x1)), where EXP and LOG are functions described in 16.10.

Except for a value raised to an integer power, Tables 2 and 3 specify that if two operands are of different type, the operand that differs in type from the result of the operation is converted to the type of the result and then the operator operates on a pair of operands of the same type. When a primary of type real, double precision, half precision, or complex is raised to an integer power, the integer operand need not be converted. If the value of I2 is negative, the interpretation of I1**I2 is the same as the interpretation of 1/(I1**ABS(I2)), which is subject to the rules for integer division (6.1.5). For example, 2**(-3) has the value of 1/(2**3), which is zero.

The type and interpretation of an expression that consists of an operator operating on either a single operand or a pair of operands are independent of the context in which the expression appears. In particular, the type and interpretation of such an expression are independent of the type of any other operand of any larger expression in which it appears. For example, if X is of type real, J is of type integer, and INT is the real-to-integer conversion function, the expression INT(X+J) is an integer expression and X+J is a real expression.

6.1.4.1 Boolean_Operands_and_Arithmetic_Operators

A Boolean operand (either base or power) of the operator ** is converted to integer and the operation is performed on the converted operand. A Boolean operand of the operator +, -, * or / is subject to the following rules:

> If two operands are of different type and one type is Boolean, the result has the type of the other operand. If both operands are of type Boolean, the result has type integer. The result of the operator + or the operator - operating on a single Boolean operand is of type integer. A Boolean operand is converted to the type of the result, and the operation is performed on the converted operand.

6.1.5 INTEGER_DIVISION.

One operand of type integer may be divided by another operand of type integer. Although the mathematical quotient of two integers is not necessarily an integer, Table 2 specifies that an expression involving the division operator with two operands of type integer is interpreted as an expression of type integer. The result of such a division is called an integer quotient and is obtained as follows: If the magnitude of the mathematical quotient is less than one, the integer quotient is zero. Otherwise, the integer quotient is the

integer whose magnitude is the largest integer that does not exceed the magnitude of the mathematical quotient and whose sign is the same as the sign of the mathematical quotient. For example, the value of the expression (-8)/3 is (-2).

6.2 CHARACTER EXPRESSIONS

A character expression is used to express a character string. Evaluation of a character expression produces a result of type character.

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

6.2.1 CHARACTER OPERATOR.

The character operator is:

Operator	Representing
//	Concatenation

The interpretation of the expression formed with the character operator is:

Use of Operator	Interpretation
x1 // x2	Concatenate x1 with x2

where: x1 denotes the operand to the left of the operator

x2 denotes the operand to the right of the operator

The result of a concatenation operation is a character string whose value is the value of x1 concatenated on the right with the value of x2 and whose length is the sum of the lengths of x1 and x2. For example, the value of 'AB' // 'CDE' is the string ABCDE.

6.2.2 FORM AND INTERPRETATION OF CHARACTER EXPRESSIONS.

A character expression and the operands of a character expression must identify values of type character. Except in a character

assignment statement (10.4), a character expression must not involve concatenation of an operand whose length specification is an asterisk in parentheses (8.4.2) unless the operand is the symbolic name of a constant.

6.2.2.1 Character Primaries.

The character primaries are:

(1) Character constant (4.8.1)

(2) Symbolic name of a character constant (8.6)

(3) Character variable reference (2.5)

(4) Character array element reference (5.4)

(5) Character array reference

(6) Character array section reference

(7) Character substring reference (5.10))

(8) Character function reference (16.2)

(9) Character expression enclosed in parentheses (6.2.2.2)

6.2.2.2 Character Expression.

The forms of a character expression are:

(1) Character primary

(2) Character expression // character primary

Thus, a character expression is a sequence of one or more character primaries separated by the concatenation operator. Form (2) indicates that in a character expression containing two or more concatenation operators, the primaries are combined from left to right to establish the interpretation of the expression. For example, the formation rules specify that the interpretation of the character expression

```
'AB' // 'CD' // 'EF'
```

is the same as the interpretation of the character expression

```
('AB' // 'CD') // 'EF'
```

The value of the character expression in this example is the same as that of the constant 'ABCDEF'. Note that parentheses have no effect

on the value of a character expression.

6.2.3 CHARACTER CONSTANT EXPRESSION.

A *character constant expression* is a character expression in which each primary is a character constant, the symbolic name of a character constant, or a character constant expression enclosed in parentheses. Note that variable, array element, substring, and function references are not allowed.

An *extended* character constant expression is a character constant expression except:

(1) selected elemental intrinsic functions are allowed, when referenced with constant arguments,

(2) selected array-valued intrinsic functions are allowed.

The list of allowed elemental and array-valued intrinsic functions is implementation dependant.

6.3 RELATIONAL EXPRESSIONS

A relational expression is used to compare the values of two arithmetic or Boolean expressions, two character expressions, or two bit expressions. A relational expression may not be used to compare the value of an arithmetic or Boolean expression with the value of a character expression or bit expression. A relational expression may not be used to compare the value of a bit expression with the value of a character expression.

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.

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

6.3.2 ARITHMETIC RELATIONAL EXPRESSION.

The form of an *arithmetic relational expression* is:

e1 relop e2

where: *e1* and *e2* are each an arithmetic expression, Boolean primary, or arithmetic array expression

relop is a relational operator

A complex operand is permitted only when the relational operator is .EQ. or .NE.

6.3.3 INTERPRETATION OF ARITHMETIC RELATIONAL EXPRESSIONS

An arithmetic relational expression is interpreted as having the logical value true if the values of the operands satisfy the relation specified by the operator. An arithmetic relational expression is interpreted as having the logical value false if the values of the operands do not satisfy the relation specified by the operator.

If the two operands are of different types or are both of Boolean type, the value of the relational expression

e1 relop e2

is the value of the expression

((*e1*) - (*e2*)) *relop* 0

where 0 (zero) is of the same type as the expression ((*e1*)- (*e2*)), and *relop* is the same relational operator in both expressions.

6.3.4 CHARACTER RELATIONAL EXPRESSION.

The form of a *character relational expression* is:

e1 relop e2

where: *e1* and *e2* are character expressions or character array expressions

relop is a relational operator

6.3.5 INTERPRETATION OF CHARACTER RELATIONAL EXPRESSIONS

A character relational expression is interpreted as the logical value true if the values of the operands satisfy the relation specified by the operator. A character relational expression is

interpreted as the logical value false if the values of the operands do not satisfy the relation specified by the operator.

The character expression e1 is considered to be less than e2 if the value of e1 precedes the value of e2 in the collating sequence; e1 is greater than e2 if the value of e1 follows the value of e2 in the collating sequence (3.1.5). Note that the collating sequence depends on the collation weight table used by the processor; however, the result of the use of the operators .EQ. and .NE. can depend on the collating sequence only if the user-specified collation weight table is selected, and then only if the program has associated the same weight with two or more characters (3.1.5).

A character relational expression that is a logical primary of a logical constant expression appearing in a nonexecutable statement (7.2) or a conditional compilation IF-directive (3.7.2) is interpreted using the ASCII collating sequence (Appendix A). Use of the C$ collation control directive (3.7.3) does not affect the interpretation of such expressions.

6.3.6 BIT RELATIONAL EXPRESSION

The form of a bit relational expression is:

e1 relop e2

where: e1 and e2 are bit expressions or bit array expressions

relop is a relational operator

6.3.7 INTERPRETATION OF BIT RELATIONAL EXPRESSIONS

A bit relational expression is interpreted as the logical value true if the values of the operands satisfy the relation specified by the operator. A bit relational expression is interpreted as the logical value false if the values of the operands do not satisfy the relation specified by the operator.

6.4 LOGICAL EXPRESSIONS

A logical expression is used to express a logical computation. Evaluation of a logical expression produces a result of type logical, with a value of true or false.

The simplest form of a logical expression is a logical constant, symbolic name of a logical constant, logical variable reference, logical array element reference, logical array reference, logical array section reference, logical function reference, or relational expression. More complicated logical expressions may be formed by using one or more logical operands together with logical operators

and parentheses.

6.4.1 LOGICAL OPERATORS.

The *logical operators* are:

Operator	Representing
.NOT.	Logical Negation
.AND.	Logical Conjunction
.OR.	Logical Inclusive Disjunction
.EQV.	Logical Equivalence
.NEQV. or .XOR.	Logical Non-Equivalence

6.4.2 FORM AND INTERPRETATION OF LOGICAL EXPRESSIONS.

A set of formation rules is used to establish the interpretation of a logical expression that contains two or more logical operators. There is a precedence among the logical operators, which determines the order in which the operands are to be combined unless the order is changed by the use of parentheses. The precedence of the logical operators is as follows:

Operator	Precedence
.NOT.	Highest
.AND.	
.OR.	
.EQV., .NEQV. or .XOR.	Lowest

For example, in the expression

```
A .OR. B .AND. C
```

the .AND. operator has higher precedence than the .OR. operator; therefore, the interpretation of the above expression is the same as the interpretation of the expression

```
A .OR. (B .AND. C)
```

The *logical operands* are:

(1) Logical primary

(2) Logical factor

(3) Logical term

(4) Logical disjunct

(5) Logical expression

The formation rules to be applied in establishing the interpretation of a logical expression are in 6.4.2.1 through 6.4.2.5.

6.4.2.1 Logical Primaries.

The logical primaries are:

(1) Logical constant (4.7.1)

(2) Symbolic name of a logical constant (8.6)

(3) Logical variable reference (2.5)

(4) Logical array element reference (5.4)

(5) Logical array reference

(6) Logical array section reference

(7) Logical function reference (16.2)

(8) Relational expression (6.3)

(9) Logical expression enclosed in parentheses (6.4.2.5)

6.4.2.2 Logical Factor.

The forms of a logical factor are:

(1) Logical primary

(2) .NOT. logical primary

6.4.2.3 Logical Term.

The forms of a logical term are:

(1) Logical factor

(2) Logical term .AND. logical factor

Thus, a logical term is a sequence of logical factors seperated by the .AND. operator. Form (2) indicates that in interpreting a logical term containing two or more .AND. operators, the logical factors are combined from left to right.

6.4.2.4 Logical Disjunct.

The forms of a logical disjunct are:

(1) Logical term

(2) Logical disjunct .OR. logical term

Thus, a logical disjunct is a sequence of logical terms separated by the .OR. operator. Form (2) indicates that in interpreting a logical disjunct containing two or more .OR. operators, the logical terms are combined from left to right.

6.4.2.5 Logical Expression.

The forms of a logical expression are:

(1) Logical disjunct

(2) Logical expression .EQV. logical disjunct

(3) Logical expression .NEQV. logical disjunct

(4) Logical expression .XOR. logical disjunct

Thus, a logical expression is a sequence of logical disjuncts separated by either the .EQV., .NEQV., or .XOR. operator. Forms (2), (3), and (4) indicate that in interpreting a logical expression containing two or more .EQV., .NEQV.,or .XOR. operators, the logical disjuncts are combined from left to right.

6.4.3 VALUE OF LOGICAL FACTORS, TERMS, AND EXPRESSIONS

The value of a logical factor involving .NOT. is shown below:

x2	.NOT. x2
true	false
false	true

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

x1	x2	x1 .AND. x2
true	true	true
true	false	false
false	true	false
false	false	false

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

x1	x2	x1 .OR. x2
true	true	true
true	false	true
false	true	true
false	false	false

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

x1	x2	x1 .EQV. x2
true	true	true
true	false	false
false	true	false
false	false	true

The value of a logical expression involving .NEQV. or .XOR. is shown below:

x1 x1	x2 x2	x1 .NEQV. x2 x1 .XOR. x2
true	true	false
true	false	true
false	true	true
false	false	false

6.4.4 <u>LOGICAL CONSTANT EXPRESSION</u>.

A <u>logical constant expression</u> is a logical expression in which each primary is a logical constant, the symbolic name of a logical constant, a relational expression in which each primary is a constant expression, or a logical constant expression enclosed in parentheses. Note that variable, array element, and function

references are not allowed.

An *extended* logical constant expression is a logical constant expression except:

(1) selected elemental intrinsic functions are allowed, when referenced with constant arguments,

(2) selected array-valued intrinsic functions are allowed.

The list of allowed elemental and array-valued intrinsic functions is implementation dependant.

6.5 PRECEDENCE OF OPERATORS

In 6.1.2 and 6.4.2 precedences have been established among the arithmetic operators and the logical operators, respectively. There is only one character operator. No precedence has been established among the relational operators. The precedences among the various operators are:

Operator	Precedence
Arithmetic	Highest
Character	
Relational	
Bit	
Logical	Lowest

An expression may contain more than one kind of operator. For example, the logical expression

```
L .OR. A + B .GE. C
```

where A, B, and C are of type real and L is of type logical, contains an arithmetic operator, a relational operator, and a logical operator. This expression would be interpreted the same as the expression

```
L .OR. ((A + B) .GE. C)
```

6.5.1 SUMMARY OF INTERPRETATION RULES.

The order in which primaries are combined using operators is determined by the following:

(1) Use of parentheses

(2) Precedence of the operators

(3) Right-to-left interpretation of exponentiations in a factor

(4) Left-to-right interpretation of multiplications and divisions in a term

(5) Left-to-right interpretation of additions and subtractions in an arithmetic expression

(6) Left-to-right interpretation of concatenations in a character expression

(7) Left-to-right interpretation of conjunctions in a logical term, Boolean term, or a bit term

(8) Left-to-right interpretation of disjunctions in a logical disjunct, Boolean disjunct, or a bit disjunct

(9) Left-to-right interpretation of equivalences in a logical expression, Boolean expression, or bit expression

6.6 EVALUATION OF EXPRESSIONS

This section applies to arithmetic, character, relational, bit, Boolean, and logical expressions.

Any variable, array, array section, array element, function, or character substring referenced as an operand in an expression must be defined at the time the reference is executed. An integer operand must be defined with an integer value, rather than a statement label value. Note that if a character string or substring is referenced, all of the referenced characters must be defined at the time the reference is executed.

Any arithmetic operation whose result is not mathematically defined is prohibited in the execution of an executable program. Examples are dividing by zero and raising a zero- valued primary to a zero-valued or negative-valued power. Raising a negative-valued primary to a real or double precision power is also prohibited.

The execution of a function reference in a statement may not alter the value of any other entity within the statement in which the function reference appears. The execution of a function reference in a statement may not alter the value of any entity in common (8.3) that affects the value of any other function reference in that statement. However, execution of a function reference in the expression _e_ of a logical IF statement (11.5) is permitted to affect entities in the statement _st_ that is executed when the value of the expression _e_ is true. If a function reference causes definition of an actual argument of the function, that argument or any associated entities must not appear elsewhere in the same statement. For example, the statements

```
A(I) = F(I)

Y = G(X) + X
```

are prohibited if the reference to F defines I or the reference to G defines X.

The data type of an expression in which a function reference appears does not affect the evaluation of the actual arguments of the function. The data type of an expression in which a function reference appears is not affected by the evaluation of the actual arguments of the function, except that the result of a generic function reference assumes a data type that depends on the data type of its arguments as specified in 16.10.

Any execution of an array element reference requires the evaluation of its subscript. The data type of an expression in which a subscript appears does not affect, nor is it affected by, the evaluation of the subscript.

Any execution of a substring reference requires the evaluation of its substring expressions. The data type of an expression in which a substring name appears does not affect, nor is it affected by, the evaluation of the substring expressions.

Any execution of an array section reference requires the evaluation of its section subscript expressions. The data type of an expression in which an array section appears does not affect, nor is it affected by, the evaluation of the array section.

When an arithmetic operator, character operator, relational operator, or logical operator operates on a pair of operands and at least one of the operands is an array operand, the operands must be conformable. The arithmetic operation, character operation, relational operation, or logical operation is performed element-by-element on corresponding array elements of the operands. The result of the operation is the same shape as the array operand or array operands. For example, the array expression

```
A+B
```

produces an array the same shape as A and B. The individual array elements of the result have the values of the first element of A added to the first element of B, the second element of A added to the second element of B, etc. The processor may perform the element-by-element operations in any order it chooses.

When an arithmetic operator or a logical operator operates on a single array operand, the operation is performed element-by-element and the result is the same shape as the operand.

6.6.1 EVALUATION OF OPERANDS.

It is not necessary for a processor to evaluate all of the operands of an expression if the value of the expression can be determined otherwise. This principle is most often applicable to logical expressions and zero-sized arrays, but it applies to all expressions. For example, in evaluating the logical expression

```
X .GT. Y .OR. L(Z)
```

where X, Y, and Z are real and L is a logical function, the function reference L(Z) need not be evaluated if X is greater than Y. Similarly in the array expression

```
X + Y(Z)
```

where X is of size zero and Y is an array-valued function, the function reference Y(Z) need not be evaluated. If a statement contains a function reference in a part of an expression that need not be evaluated, all entities that would have become defined in the execution of that reference become undefined at the completion of evaluation of the expression containing the function reference. In the example above, evaluation of the expression causes Z to become undefined if L defines its argument.

6.6.2 ORDER OF EVALUATION OF FUNCTIONS.

If a statement contains more than one function reference, a processor may evaluate the functions in any order, except for a logical IF statement and a function argument list containing function references. For example, the statement

```
Y = F(G(X))
```

where F and G are functions, requires G to be evaluated before F is evaluated.

In a statement that contains more than one function reference, the value provided by each function reference must be independent of the order chosen by the processor for evaluation of the function references.

6.6.3 INTEGRITY OF PARENTHESES.

The sections that follow state certain conditions under which a processor may evaluate an expression different from the one obtained by applying the interpretation rules given in 6.1 through 6.5. However, any expression contained in parentheses must be treated as an entity. For example, in evaluating the expression A*(B*C), the product of B and C must be evaluated and then multiplied by A; the processor must not evaluate the mathematically equivalent expression

(A*B)*C.

6.6.4 RESTRICTIONS ON APPEARANCE OF ARRAY EXPRESSIONS

Array-valued expressions may not appear:

(1) As a logical expression of a logical IF statement

(2) As a logical expression of a block IF or ELSE IF statement

(3) As e1, e2, or e3 expressions of a DO statement

(4) As an arithmetic expression in an arithmetic IF statement

(5) In a computed GO TO, OPEN, CLOSE, INQUIRE, or RETURN statement

(6) In the REC= specifier of a cilist

(7) In a substring expression

(8) As an external unit identifier

6.6.5 EVALUATION OF ARITHMETIC EXPRESSIONS

The rules given in 6.1.2 specify the interpretation of an arithmetic expression. Once the interpretation has been established in accordance with those rules, the processor may evaluate any mathematically equivalent expression, provided that the integrity of parentheses is not violated.

Two arithmetic expressions are mathematically equivalent if, for all possible values of their primaries and Boolean primaries (6.7.1.1), their mathematical values are equal. However, mathematically equivalent arithmetic expressions may produce different computational results.

The mathematical definition of integer division is given in 6.1.5. The difference between the value of the expression 5/2 and 5./2. is a mathematical difference, not a computational difference. If Boolean operands (6.7.1) are present in an arithmetic expression, its operators may not have the associative and distributive properties that would yield mathematically equivalent expressions if all operands were arithmetic. For example, the expressions

```
2.0 + O"1" + O"1" and
2.0 + (O"1" + O"1")
```

are not mathematically equivalent.

The following are examples of expressions, along with allowable alternative forms that may be used by the processor in the evaluation of those expressions. A, B, and C represent arbitrary real, double precision, or complex operands; I and J represent arbitrary integer operands; and X, Y, and Z represent arbitrary arithmetic operands.

Expression	Allowable Alternative Form
X+Y	Y+X
X*Y	Y*X
-X+Y	Y-X
X+Y+Z	X+(Y+Z)
X-Y+Z	X-(Y-Z)
X*B/Z	X*(B/Z)
X*Y-X*Z	X*(Y-Z)
A/B/C	A/(B*C)
A/5.0	0.2*A

The following are examples of expressions along with forbidden forms that must not be used by the processor in the evaluation of those expressions.

Expression	Non-Allowable Alternative Form
I/2	0.5*I
X*I/J	X*(I/J)
I/J/A	I/(J*A)
(X*Y)-(X*Z)	X*(Y-Z)
X*(Y-Z)	X*Y-X*Z

In addition to the parentheses required to establish the desired interpretation, parentheses may be included to restrict the alternative forms that may be used by the processor in the actual evaluation of the expression. This is useful for controlling the magnitude and accuracy of intermediate values developed during the evaluation of an expression. For example, in the expression

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

the term (B-C) must be evaluated and then added to A. Note that the inclusion of parentheses may change the mathematical value of an expression. For example, the two expressions:

```
A*I/J
```

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

may have different mathematical values if I and J are factors of integer data type.

Each operand of an arithmetic operator has a data type that may depend on the order of evaluation used by the processor. For example, in the evaluation of the expression

```
D+R+I
```

where D, R, and I represent 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) is added first.

6.6.6 EVALUATION OF CHARACTER EXPRESSIONS

The rules given in 6.2.2 specify the interpretation of a character expression as a string of characters. A processor needs to evaluate only as much of the character expression as is required by the context in which the expression appears. For example, the statements

```
CHARACTER*2 C1,C2,C3,CF
C1 = C2 // CF(C3)
```

do not require the function CF to be evaluated, because only the value of C2 is needed to determine the value of C1.

6.6.7 EVALUATION OF RELATIONAL EXPRESSIONS

The rules given in 6.3.3 and 6.3.5 specify the interpretation of relational expressions. Once the interpretation of an expression has been established in accordance with those rules, the processor may evaluate any other expression that is relationally equivalent. For example, the processor may choose to evaluate the relational expression

```
I .GT. J
```

where I and J are integer variables, as

```
J - I .LT. 0
```

Two relational expressions are relationally equivalent if their logical values are equal for all possible values of their primaries.

6.6.8 EVALUATION OF LOGICAL EXPRESSIONS

The rules given in 6.4.2 specify the interpretation of a logical expression. Once the interpretation of an expression has been established in accordance with those rules, the processor may evaluate any other expression that is logically equivalent, provided that the integrity of parentheses is not violated. For example, the processor may choose to evaluate the logical expression

```
L1 .AND. L2 .AND. L3
```

where L1, L2, and L3 are logical variables, as

```
L1 .AND. (L2 .AND. L3)
```

Two logical expressions are logically equivalent if their values are equal for all possible values of their primaries.

6.7 BOOLEAN EXPRESSIONS

A Boolean expression is formed with logical operators and Boolean operands and/or arithmetic operands (6.1.2). Evaluation of a Boolean expression produces a result of type Boolean.

6.7.1 BOOLEAN OPERANDS

The Boolean operands are:

(1) Boolean primary

(2) Boolean factor

(3) Boolean term

(4) Boolean disjunct

(5) Boolean expression

The formation rules to be applied in establishing the interpretation of a Boolean expression are in Sections 6.7.1.1 through 6.7.1.5.

6.7.1.1 Boolean Primary

The Boolean primaries are:

(1) Unsigned Boolean constant (4.9.1)

(2) Symbolic name of a Boolean constant (8.6)

(3) Boolean variable reference (2.5)

(4) Boolean array element reference (5.4)

(5) Boolean function reference (16.2)

(6) Boolean array reference

(7) Boolean array section reference

(8) Boolean expression enclosed in parentheses (6.7.1.5)

6.7.1.2 Boolean Factor

The forms of a Boolean_factor are

(1) Boolean primary

(2) .NOT. Boolean primary

(3) .NOT. arithmetic expression

6.7.1.3 Boolean Term

The forms of a Boolean_term are

(1) Boolean factor

(2) Boolean term .AND. Boolean factor

(3) Boolean term .AND. arithmetic expression

(4) Arithmetic expression .AND. Boolean factor

(5) Arithmetic expression .AND. arithmetic expression

Thus a Boolean term is a sequence of Boolean factors and/or arithmetic expressions, separated by the .AND. operator. Forms (2) and (3) indicate that in interpreting a Boolean term containing two or more .AND. operators, the Boolean factors and arithmetic expressions are combined from left to right.

6.7.1.4 Boolean Disjunct

The forms of a Boolean_disjunct are:

(1) Boolean term

(2) Boolean disjunct .OR. Boolean term

(3) Boolean disjunct .OR. arithmetic expression

(4) Arithmetic expression .OR. Boolean term

(5) Arithmetic expression .OR. arithmetic expression

Thus a Boolean disjunct is a sequence of Boolean terms and/or arithmetic expressions, separated by the .OR. operator. Forms (2) and (3) indicate that in interpreting a Boolean disjunct containing two or more .OR. operators, the Boolean terms and arithmetic expressions are combined from left to right.

6.7.1.5 Boolean Expression

The forms of a Boolean expression are:

(1) Boolean disjunct

(2) Boolean expression .EQV. Boolean disjunct

(3) Boolean expression .EQV. arithmetic expression

(4) Arithmetic expression .EQV. Boolean disjunct

(5) Arithmetic expression .EQV. arithmetic expression

(6) Boolean expression .NEQV. Boolean disjunct

(7) Boolean expression .NEQV. arithmetic expression

(8) Arithmetic expression .NEQV. Boolean disjunct

(9) Arithmetic expression .NEQV. arithmetic expression

(10) Boolean expression .XOR. Boolean disjunct

(11) Boolean expression .XOR. arithmetic expression

(12) Arithmetic expression .XOR. Boolean disjunct

(13) Arithmetic expression .XOR. arithmetic expression

Thus, a Boolean expression is a sequence of Boolean disjuncts separated by the .EQV., .NEQV., or .XOR. operators. Forms (2), (3), (6), (7), (10), and (11) indicate that in interpreting a Boolean expression containing two or more .EQV., .NEQV., or .XOR. operators, the Boolean disjuncts and arithmetic expressions are combined from left to right.

6.7.2 VALUE OF BOOLEAN FACTORS, TERMS, AND EXPRESSIONS

If an operand is of type integer, real, double precision, half precision, or complex, it is converted to Boolean and the operation performed on the converted operand. Conversion to Boolean is by means of the generic function BOOL (16.10).

A Boolean operator determines each bit value of the value it yields independently of other bits of the value. Each bit value is determined from the corresponding bit values of the operand(s). (Two bit positions correspond if they have the same ordinal in their storage sequence within their respective storage units.)

Each bit value of a Boolean factor involving .NOT. is shown below:

X2	.NOT. X2
0	1
1	0

Each bit value of a Boolean term involving .AND. is shown below:

X1	X2	X1 .AND. X2
0	0	0
0	1	0
1	0	0
1	1	1

Each bit value of a Boolean expression involving .OR. is shown below

X1	X2	X1 .OR. X2
0	0	0
0	1	1
1	0	1
1	1	1

Each bit value of a Boolean expression involving .EQV. is shown below:

X1	X2	X1 .EQV. X2
0	0	1
0	1	0
1	0	0
1	1	1

Each bit value of a Boolean expression involving .NEQV. (or .XOR.) is shown below:

X1	X2	X1 .NEQV. X2
0	0	0
0	1	1
1	0	1
1	1	0

6.7.3 BOOLEAN CONSTANT EXPRESSION

A Boolean constant expression is a Boolean expression in which each Boolean primary is a Boolean constant, the symbolic name of a Boolean constant, or a Boolean constant expression enclosed in parentheses, and each arithmetic primary is an arithmetic constant, the symbolic name of an arithmetic constant, or an arithmetic constant expression enclosed in parentheses.

An extended Boolean constant expression is a Boolean constant expression except:

(1) selected elemental intrinsic functions are allowed, when referenced with constant arguments,

(2) selected array-valued intrinsic functions are allowed.

The list of allowed elemental and array-valued intrinsic functions is implementation dependant.

6.8 CONSTANT EXPRESSIONS

A constant expression is an arithmetic constant expression (6.1.3), a character constant expression (6.2.3), a bit constant expression (6.9.3), a logical constant expression (6.4.4), a Boolean constant expression (6.7.3), or a bit constant expression (6.9.3).

An *extended* constant expression is a constant expression except:

(1) selected elemental intrinsic functions are allowed, when referenced with constant arguments,

(2) selected array-valued intrinsic functions are allowed.

The list of allowed elemental and array-valued intrinsic functions is implementation dependant.

In order to eliminate circularity of definition, no variable or array name may be referenced in an extended constant expression unless that variable or array has been defined in preceding specification statement(s). For expample,

```
REAL A( SIZE(B))
REAL B( SIZE(A))
```

is prohibited because B was referenced prior to its definition.

6.9 BIT EXPRESSIONS

A bit expression is used to express a bit computation. Evaluation of a bit expression produces a result of type bit, with a value of B"1" or B"0".

The simplest form of a bit expression is a bit constant, symbolic name of a bit constant, bit variable reference, bit array element reference, bit array reference, bit array section reference, or bit function reference. More complicated bit expressions may be formed by using one or more bit operands together with bit operators and parentheses.

6.9.1 BIT OPERATORS.

The *bit operators* are:

Operator	Representing
.BNOT.	Bit Negation
.BAND.	Bit Conjunction
.BOR.	Bit Inclusive Disjunction
.BEQV.	Bit Equivalence
.BNEQV. or .BXOR.	Bit Non-Equivalence

6.9.2 FORM AND INTERPRETATION OF BIT EXPRESSIONS.

A set of formation rules is used to establish the interpretation of a bit expression that contains two or more bit operators. There is a precedence among the bit operators, which determines the order in which the operands are to be combined unless the order is changed by the use of parentheses. The precedence of the bit operators is as follows:

Operator	Precedence
.BNOT.	Highest
.BAND.	
.BOR.	
.BEQV., .BNEQV. or .BXOR.	Lowest

For example, in the expression

```
A .BOR. B .BAND. C
```

the .BAND. operator has higher precedence than the .BOR. operator; therefore, the interpretation of the above expression is the same as the interpretation of the expression

```
A .BOR. (B .BAND. C)
```

The bit operands are:

(1) Bit primary

(2) Bit factor

(3) Bit term

(4) Bit disjunct

(5) Bit expression

The formation rules to be applied in establishing the interpretation of a bit expression are in 6.9.2.1 through 6.9.2.5.

6.9.2.1 Bit Primaries.

The bit primaries are:

(1) Bit constant (4.11.1)

(2) Symbolic name of a bit constant (8.6)

(3) Bit variable reference (2.5)

(4) Bit array element reference (5.4)

(5) Bit array reference

(6) Bit array section reference

(7) Bit function reference (16.2)

(8) Bit expression enclosed in parentheses (6.9.2.5)

6.9.2.2 Bit_Factor.

The forms of a bit factor are:

(1) Bit primary

(2) .BNOT. bit primary

6.9.2.3 Bit_Term.

The forms of a bit term are:

(1) Bit factor

(2) Bit term .BAND. bit factor

Thus, a bit term is a sequence of bit factors separated by the .BAND. operator. Form (2) indicates that in interpreting a bit term containing two or more .BAND. operators, the bit factors are combined from left to right.

6.9.2.4 Bit_Disjunct.

The forms of a bit disjunct are:

(1) Bit term

(2) Bit disjunct .BOR. bit term

Thus, a bit disjunct is a sequence of bit terms separated by the .BOR. operator. Form (2) indicates that in interpreting a bit disjunct containing two or more .BOR. operators, the bit terms are combined from left to right.

6.9.2.5 Bit_Expression.

The forms of a bit expression are:

(1) Bit disjunct

(2) Bit expression .BEQV. bit disjunct

(3) Bit expression .BNEQV. bit disjunct

(4) Bit expression .BXOR. bit disjunct

Thus, a bit expression is a sequence of bit disjuncts separated by either the .BEQV., .BNEQV., or .BXOR. operator. Forms (2), (3), and (4) indicate that in interpreting a bit expression containing two or more .BEQV., .BNEQV.,or .BXOR. operators, the bit disjuncts are combined from left to right.

6.9.3 VALUE OF BIT FACTORS, TERMS, AND EXPRESSIONS

The value of a bit factor involving .BNOT. is shown below:

x2	.BNOT. x2
B"1"	B"0"
B"0"	B"1"

The value of a bit term involving .BAND. is shown below:

x1	x2	x1 .BAND. x2
B"1"	B"1"	B"1"
B"1"	B"0"	B"0"
B"0"	B"1"	B"0"
B"0"	B"0"	B"0"

The value of a bit disjunct involving .BOR. is shown below:

x1	x2	x1 .BOR. x2
B"1"	B"1"	B"1"
B"1"	B"0"	B"1"
B"0"	B"1"	B"1"
B"0"	B"0"	B"0"

The value of a bit expression involving .BEQV. is shown below:

x1	x2	x1 .BEQV. x2
B"1"	B"1"	B"1"
B"1"	B"0"	B"0"
B"0"	B"1"	B"0"
B"0"	B"0"	B"1"

The value of a bit expression involving .BNEQV. or .BXOR. is shown below:

x1 x1	x2 x2	x1 .BNEQV. x2 x1 .BXOR. x2
B"1"	B"1"	B"0"
B"1"	B"0"	B"1"
B"0"	B"1"	B"1"
B"0"	B"0"	B"0"

6.9.4 BIT CONSTANT EXPRESSION.

A *bit constant expression* is a bit expression in which each primary is a bit constant, the symbolic name of a bit constant, or a bit constant expression enclosed in parentheses. Note that variable, array element, and function references are not allowed.

An *extended* bit constant expression is a bit constant expression except:

(1) selected elemental intrinsic functions are allowed, when referenced with constant arguments,

(2) selected array-valued intrinsic functions are allowed.

The list of allowed elemental and array-valued intrinsic functions is implementation dependant.

7.0 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 specify characteristics, arrangement, and initial values of data; contain editing information; specify statement functions; classify 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 must not be used to control the execution sequence.

7.1 EXECUTABLE STATEMENTS

The following statements are classified as executable:

(1) Arithmetic, logical, statement label (ASSIGN), bit, Boolean, and character assignment statements

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

(3) Arithmetic IF and logical IF statements

(4) Block IF, ELSE IF, ELSE, and END IF statements

(5) CONTINUE statement

(6) STOP and PAUSE statements

(7) DO statement

(8) READ, WRITE, PRINT, PUNCH, ENCODE, DECODE, BUFFER IN, and BUFFER OUT statements

(9) REWIND, BACKSPACE, ENDFILE, OPEN, CLOSE, and INQUIRE statements

(10) CALL and RETURN statements

(11) ALLOCATE and FREE statements

(12) Logical WHERE, Block WHERE, OTHERWISE, and ENDWHERE statements

(13) IDENTIFY statement

(14) FORALL statement

(15) END statement

7.2 NONEXECUTABLE STATEMENTS

The following statements are classified as nonexecutable:

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

(2) DIMENSION, COMMON, EQUIVALENCE, IMPLICIT, PARAMETER, EXTERNAL, INTRINSIC, and SAVE statements

(3) INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, BIT, HALF PRECISION, BOOLEAN, and CHARACTER type-statements

(4) DATA statement

(5) FORMAT statement

(6) Statement function statement

(7) NAMELIST statement

(8) ROWWISE statement

(9) INTERFACE and END INTERFACE statements

8.0 SPECIFICATION STATEMENTS

There are twelve kinds of specification statements:

(1) DIMENSION

(2) EQUIVALENCE

(3) COMMON

(4) INTEGER, REAL, DOUBLE PRECISION, HALF PRECISION, COMPLEX, LOGICAL, BIT, BOOLEAN, and CHARACTER type-statements

(5) IMPLICIT

(6) PARAMETER

(7) EXTERNAL

(8) INTRINSIC

(9) SAVE

(10) ROWWISE

(11) Procedure interface information, INTERFACE and END INTERFACE

(12) VIRTUAL

All specification statements are nonexecutable.

8.1 DIMENSION STATEMENT

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

The form of a DIMENSION statement is:

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

where each a(d) is an array declarator (5.2).

Each symbolic name a appearing in a DIMENSION statement declares a to be an array 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 name in an array declarator in a program unit is permitted.

8.2 EQUIVALENCE STATEMENT

An EQUIVALENCE statement is used to specify the sharing of storage units by two or more entities in a program unit. This causes association of the entities that share the storage units.

If the equivalenced entities are of different data types, the EQUIVALENCE statement does not cause type conversion or imply mathematical equivalence. If a variable and an array are equivalenced, the variable does not have array properties and the array does not have the properties of a variable.

8.2.1 FORM OF AN EQUIVALENCE STATEMENT

The form of an EQUIVALENCE statement is:

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

where each *nlist* is a list (2.11) of variable names, array element names, array names, and character substring names. Each list must contain at least two names. Names of dummy arguments of an external procedure in a subprogram and allocatable arrays must not appear in the list. If a variable name or array name is also a function name (16.5.1), that name must not appear in the list.

Each subscript expression or substring expression in a list *nlist* must be an extended integer constant expression.

8.2.2 EQUIVALENCE ASSOCIATION.

An EQUIVALENCE statement specifies that the storage sequences of the entities whose names appear in a list *nlist* share the same first storage unit. This causes the association of the entities in the list *nlist* and may cause association of other entities (18.1).

8.2.3 EQUIVALENCE OF ENTITIES OF DIFFERENT TYPES

An entity of any type may be equivalenced with entities of other types. The lengths of the equivalenced entities are not required to be the same. However, alignments, as specified by the user, must conform with the alignments as shown in the following table, Table 6:

Table 6
ALIGNMENT REQUIREMENTS FOR
EQUIVALENCE(X1,X2)

x1 \ x2	Integer, Real, Double precision, Complex, Logical, Boolean	Half precision	Character	Bit
Integer, Real, Double precision, Complex, Logical, Boolean	Numeric	Numeric	Numeric	Numeric
Half precision	Numeric	Half-numeric	Half-numeric	Half-numeric
Character	Numeric	Half-numeric	Character	Character
Bit	Numeric	Half-numeric	Character	Bit

An EQUIVALENCE statement specifies that the storage sequences of the character entities whose names appear in a list _nlist_ share the same first character storage unit. This causes the association of the entities in the list _nlist_ and may cause association of other entities (18.1). Any adjacent characters in the associated entities may also have the same character storage unit and thus may also be associated. In the example:

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

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

```
|01|02|03|04|05|06|07|

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

8.2.4 ARRAY NAMES AND ARRAY ELEMENT NAMES.

If an array element name appears in an EQUIVALENCE statement, the number of subscript expressions must be the same as the number of dimensions specified in the array declarator for the array name.

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.

8.2.5 RESTRICTIONS ON EQUIVALENCE STATEMENTS.

An EQUIVALENCE statement must not contain virtual (IDENTIFY) array names, automatic array names, or allocatable array names.

Alignments, as specified by the user, must conform with the alignments as shown in Table 6.

An EQUIVALENCE statement must not specify that the same storage unit is to occur more than once in a storage sequence. For example,

```
DIMENSION A(2)
EQUIVALENCE (A(1),B), (A(2),B)
```

is prohibited, because it would specify the same storage unit for A(1) and A(2). An EQUIVALENCE statement must not specify that consecutive storage units are to be nonconsecutive. For example, the following is prohibited:

```
REAL A(2)
DOUBLE PRECISION D(2)
EQUIVALENCE (A(1),D(1)), (A(2),D(2))
```

8.3 COMMON STATEMENT

The COMMON statement provides a means of associating entities in different program units. This allows different program units to define and reference the same data without using arguments, and to share storage units.

For COMMON the compiler will allocate storage in conformance with requirements as specified in the following table, Table 7:

Table 7
ALIGNMENT REQUIREMENTS
FOR COMMON

TYPE	BOUNDARY
Integer Real Dble precision Complex Logical Boolean	Numeric
Half precision	Half-numeric
Character	Character
Bit	Bit

8.3.1 FORM OF A COMMON STATEMENT.

The form of a COMMON statement is:

```
COMMON [/[cb]/] nlist [[,]/[cb]/ nlist]...
```

where: cb is a common block name (19.2.1)

nlist is a list (2.11) of variable names, array names, and array declarators. Only one appearance of a symbolic name as a variable name, array name, or array declarator is permitted in all such lists within a program unit. Names of dummy arguments of an external procedure in a subprogram must not appear in the list. If a variable name is also a function name (16.5.1), that name must not appear in the list.

Each omitted cb specifies the blank common block. If the first cb is omitted, the first two slashes are optional.

In each COMMON statement, the entities whose names appear in an nlist following a block name cb are declared to be in common block cb. If the first cb is omitted, all entities whose names appear in the first nlist are specified to be in blank common. Alternatively, the appearance of two slashes with no block name between them declares the entities whose names appear in the list nlist that follows to be in blank common.

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.

8.3.2 COMMON BLOCK STORAGE SEQUENCE.

For each common block, a *common block storage sequence* is formed as follows:

(1) A storage sequence is formed consisting of the storage sequences of all entities in the lists *nlist* for the common block. The order of the storage sequence is the same as the order of the appearance of the lists *nlist* in the program unit.

(2) The storage sequence formed in (1) is extended to include all storage units of any storage sequence associated with it by equivalence association. The sequence may be extended only by adding storage units beyond the last storage unit. Entities associated with an entity in a common block are considered to be in that common block.

8.3.3 SIZE OF A COMMON BLOCK.

The *size of a common block* is the size of its common block storage sequence, including any extensions of the sequence resulting from equivalence association.

Within an executable program, all named common blocks that have the same name must be the same size. Blank common blocks within an executable program are not required to be the same size.

8.3.4 COMMON ASSOCIATION.

Within an executable program, the common block storage sequences of all common blocks with the same name have the same first storage unit. Within an executable program, the common block storage sequences of all blank common blocks have the same first storage unit. This results in the association (18.1) of entities in different program units.

8.3.5 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) Execution of a RETURN or END statement sometimes causes entities in named common blocks to become undefined but never causes entities in blank common to become undefined (16.8.4).

(2) Named common blocks of the same name must be of the same size in all program units of an executable program in which they appear, but blank common blocks may be of different sizes.

(3) Entities in named common blocks may be initially defined by means of a DATA statement in any program unit, but entities in blank common must not be initially defined (Section 9).

8.3.6 RESTRICTIONS ON COMMON AND EQUIVALENCE.

An EQUIVALENCE statement must not cause the storage sequences of two different common blocks in the same program unit to be associated. Equivalence association must not cause a common block storage sequence to be extended by adding storage units preceding the first storage unit of the first entity specified in a COMMON statement for the common block. For example, the following is not permitted:

```
COMMON /X/A
REAL B(2)
EQUIVALENCE (A,B(2))
```

Virtual (IDENTIFY) arrays, automatic arrays, and allocatable arrays may not be specified in a COMMON statement.

8.4 TYPE-STATEMENTS

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

The appearance of the symbolic name of a constant, variable, array, external function, or statement function in a type-statement specifies the data type for that name for all appearances in the program unit. Within a program unit, a name must not have its type explicitly specified more than once.

A type-statement that confirms the type of an intrinsic function whose name appears in the Specific Name column of Table 5 is not required, but is permitted. If a generic function name appears in a type-statement, such an appearance is not sufficient by itself to remove the generic properties from that function.

The name of a main program, subroutine, or block data subprogram must not appear in a type-statement.

8.4.1 BIT, BOOLEAN, AND ARITHMETIC TYPE-STATEMENTS

An INTEGER, REAL, DOUBLE PRECISION, HALF PRECISION, COMPLEX, LOGICAL, BIT, or BOOLEAN type-statement is of the form

<u>typ</u> <u>v</u> [,<u>v</u>]...

where:

<u>typ</u> is one of INTEGER, REAL, DOUBLE PRECISION, HALF PRECISION, COMPLEX, LOGICAL, BIT, or BOOLEAN

<u>v</u> is a variable name, array name, array declarator, symbolic name of a constant, function name, or dummy procedure name (19.2.11).

8.4.2 CHARACTER TYPE-STATEMENT.

The form of a CHARACTER type-statement is:

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

where: <u>nam</u> is of one of the forms:

```
v [*len]

a [(d)] [*len]
```

<u>v</u> is a variable name, symbolic name of a constant, function name, or dummy procedure name

<u>a</u> is an array name

<u>a(d)</u> is an array declarator

<u>len</u> is the length (number of characters) of a character variable, character array element, character constant that has a symbolic name, or character function, and is called the <u>length specification</u>. <u>len</u> is one of the following:

(1) An unsigned, nonzero, integer constant

(2) An extended integer constant expression (6.1.3.1) enclosed in parentheses and with a positive value

(3) An asterisk in parentheses, (*)

A length <u>len</u> immediately following the word CHARACTER is the length specification for each entity in the 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.

An entity declared in a CHARACTER statement must have a length specification that is an extended integer constant expression,

unless that entity is an external function, a dummy argument of an external procedure, or a character constant that has a symbolic name.

If a dummy argument has a len of (*) declared, the dummy argument assumes the length of the associated actual argument for each reference of the subroutine or function. If the associated actual argument is an array name, the length assumed by the dummy argument is the length of an array element in the associated actual argument array.

If an external function has a len of (*) declared in a function subprogram, the function name must appear as the name of a function in a FUNCTION or ENTRY statement in the same subprogram. When a reference to such a function is executed, the function assumes the length specified in the referencing program unit.

The length specified for a character function in the program unit that references the function must be an extended integer constant expression and must agree with the length specified in the subprogram that specifies the function. Note that there always is agreement of length if a len of (*) is specified in the subprogram that specifies the function.

If a character constant that has a symbolic name has a len of (*) declared, the constant assumes the length of its corresponding extended constant expression in a PARAMETER statement.

The length specified for a character statement function or statement function dummy argument of type character must be an extended integer constant expression.

8.5 IMPLICIT STATEMENT

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

The form of an IMPLICIT statement is:

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

where: typ is one of INTEGER, REAL, DOUBLE PRECISION, HALF PRECISION, COMPLEX, LOGICAL, BIT, BOOLEAN, or CHARACTER [*len]

a is either a single letter or a range of single letters in alphabetical order. A range is denoted by the first and last letter of the range separated by a minus. Writing a range of letters a1 - a2 has the same effect as writing a list of the single letters a1 through a2.

len is the length of the character entities and is one of the following:

(1) An unsigned, nonzero, integer constant

(2) An extended integer constant expression (6.1.3.1) enclosed in parentheses and with a positive value

If len is not specified, the length is one.

An IMPLICIT statement specifies a type for all variables, arrays, symbolic names of constants, external functions, and statement functions that begin with any letter that appears in the specification, either as a single letter or included in a range of letters. IMPLICIT statements do not change the type of any intrinsic functions. An IMPLICIT statement applies only to the program unit that contains it.

Type specification by an IMPLICIT statement may be overridden or confirmed for any particular variable, array, symbolic name of a constant, external function, or statement function 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. Note that the length is also overridden when a particular name appears in a CHARACTER or CHARACTER FUNCTION statement.

Within the specification statements of a program unit, IMPLICIT statements must precede all other specification statements except PARAMETER statements. A program unit may contain more than one IMPLICIT statement.

The same letter must not appear as a single letter, or be included in a range of letters, more than once in all of the IMPLICIT statements in a program unit.

8.6 PARAMETER STATEMENT

A PARAMETER statement is used to give a constant a symbolic name.

The form of a PARAMETER statement is:

PARAMETER (p=e [,p=e]...)

where: p is a symbolic name

e is an extended constant expression (6.7)

If the symbolic name p is of type integer, real, double precision, half precision, complex, or Boolean, the corresponding expression e must be either an extended arithmetic constant expression (6.1.3) or

an extended Boolean constant expression (6.7.3). If the symbolic name p is of type character, bit, or logical, the corresponding expression e must be an extended character constant expression (6.2.3), an extended bit constant expression, or an extended logical constant expression (6.4.4), respectively.

Each p is the symbolic name of a constant that becomes defined with the value determined from the expression e that appears on the right of the equals, in accordance with the rules for assignment statements (10.1, 10.2, and 10.4).

Any symbolic name of a constant that appears in an expression e must have been defined previously in the same or a different PARAMETER statement in the same program unit.

A symbolic name of a constant must not become defined more than once in a program unit.

If a symbolic name of a constant is not of default implied type, its type must be specified by a type-statement or IMPLICIT statement prior to its first appearance in a PARAMETER statement. If the length specified for the symbolic name of a constant of type character is not the default length of one, its length must be specified in a type-statement or IMPLICIT statement prior to the first appearance of the symbolic name of the constent. Its length must not be changed by subsequent statements including IMPLICIT statements.

Once such a symbolic name is defined, that name may appear in that program unit in any subsequent statement as a primary in an expression, in a DATA statement (9.1), or as the real or imaginary part of a complex constant. It may also appear in a C$-directive as a primary in a expression or as a parameter value. A symbolic name of a constant must not be part of a format specification.

A symbolic name in a PARAMETER statement may identify only the corresponding constant in that program unit.

8.7 EXTERNAL STATEMENT

An EXTERNAL statement is used to identify a symbolic name as representing an external procedure or dummy procedure, and to permit such a name to be used as an actual argument.

The form of an EXTERNAL statement is:

```
EXTERNAL proc [,proc]...
```

where each proc is the name of an external procedure, dummy procedure, or block data subprogram.

Appearance of a name in an EXTERNAL statement declares that name to be an external procedure name, dummy procedure name, or block data subprogram name. If an external procedure name or a dummy procedure name is used as an actual argument in a program unit, it must appear in an EXTERNAL statement in that program unit. Note that a statement function name must not appear in an EXTERNAL statement.

If an intrinsic function name appears in an EXTERNAL statement in a program unit, that name becomes the name of some external procedure and an intrinsic function of the same name is not available for reference in the program unit.

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

8.8 INTRINSIC STATEMENT

An INTRINSIC statement is used to identify a symbolic name as representing an intrinsic function (16.3). It also permits a name that represents a specific intrinsic function to be used as an actual argument.

The form of an INTRINSIC statement is:

INTRINSIC fun [,fun]...

where each fun is an intrinsic function name.

Appearance of a name in an INTRINSIC statement declares that name to be an intrinsic function name. If a specific name of an intrinsic function is used as an actual argument in a program unit, it must appear in an INTRINSIC statement in that program unit. The names of intrinsic functions for type conversion (INT, IFIX, IHINT, IDINT, FLOAT, SNGL, HALF, EXTEND, REAL, DBLE, BOOL, BTOL, LTOB, CMPLX, ICHAR, CHAR), lexical relationship (LGE, LGT, LLE, LLT) and for choosing the largest or smallest value (MAX, MAX0, AMAX1, DMAX1, AMAX0, MAX1, MIN, MIN0, AMIN1, DMIN1, AMIN0, MIN1) must not be used as actual arguments. The names of intrinsic functions for performing logical operations (AND, OR, XOR, EQV, NEQV) also must not be used as actual arguments.

The appearance of a generic function name in an INTRINSIC statement does not cause that name to lose its generic property.

Only one appearance of a symbolic name in all of the INTRINSIC statements of a program unit is permitted. Note that a symbolic name must not appear in both an EXTERNAL and an INTRINSIC statement in a program unit.

8.9 SAVE STATEMENT

A SAVE statement is used to retain the definition status of an entity after the execution of a RETURN or END statement in a subprogram. Within a function or subroutine subprogram, an entity specified by a SAVE statement does not become undefined as a result of the execution of a RETURN or END statement in the subprogram. However, such an entity in a common block may become undefined or redefined in another program unit.

The form of a SAVE statement is:

```
SAVE [a [,a]...]
```

where each 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 not permitted.

Dummy argument names, procedure names, and names of entities in a common block must not appear in a SAVE statement.

Virtual (IDENTIFY) arrays, and automatic arrays may not be specified in a SAVE statement.

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

The appearance of a common block name preceded and followed by a slash in a SAVE statement has the effect of specifying all of the entities in that common block.

If a particular common block name is specified by a SAVE statement in a subprogram of an executable program, it must be specified by a SAVE statement in every subprogram in which that common block appears.

A SAVE statement is optional in a main program and has no effect.

If a named common block is specified in a SAVE statement in a subprogram, the current values of the entities in the common block storage sequence (8.3.3) at the time a RETURN or END statement is executed are made available to the next program unit that specifies that common block name in the execution sequence of an executable program.

If a named common block is specified in the main program unit, the current values of the common block storage sequence are made available to each subprogram that specifies that named common block; a SAVE statement in the subprogram has no effect.

The definition status of each entity in the named common block

storage sequence depends on the association that has been established for the common block storage sequence (18.2 and 18.3).

If a local entity that is specified by a SAVE statement and is not in a common block is in a defined state at the time a RETURN or END statement is executed in a subprogram, that entity is defined with the same value at the next reference of that subprogram.

The execution of a RETURN statement or an END statement within a subprogram causes all entities within the subprogram to become undefined except for the following:

(1) Entities specified by SAVE statements

(2) Entities in blank common

(3) Initially defined entities that have neither been redefined nor become undefined

(4) Entities in a named common block that appears in the subprogram and appears in at least one other program unit that is referencing, either directly or indirectly, that subprogram

8.10 ROWWISE STATEMENT

An array name appearing in a ROWWISE statement declares that array to be stored in row order (5.5.3).

The form of a ROWWISE statement is:

```
ROWWISE a[,a]...
```

where each a is an array name

An array name that is declared in a ROWWISE statement must appear with an array declarator in a type specification, DIMENSION, or COMMON statement.

8.11 PROCEDURE INTERFACE INFORMATION

The form of a procedure interface information block is:

```
INTERFACE
    interface-description
    [interface-description
        .
        .
        .    ]
END INTERFACE
```

where the form of an interface-description is:

header-statement
[specification-statements]

where:

header-statement	is a SUBROUTINE or FUNCTION statement
specification-statements	describe the dummy arguments and functional result, if applicable, of the procedure

A procedure interface information block specifies information about an external subprogram. Specification of an external subprogram in an interface information block is optional except for user array-valued functions which must be specified if they are to be referenced. The specification information must include the number, order, and type of dummy arguments, whether a dummy argument is an array (whether it is rowwise or columnwise must be specified) or not. For array-valued functions, the shape of the function result must be specified.

A subroutine or function name appearing in a SUBROUTINE or FUNCTION statement of an interface description may be an external or dummy procedure in the program unit containing the interface description. If the name is an external procedure name, the interface information given by the interface description overrides (for the program unit containing it) the interface information appearing in the actual program unit defining the external procedure.

A procedure interface information block may appear where "Other Specification Statements" may appear.

Procedure interface information may be unavailable for an external procedure because the external procedure is specified by means other than a FORTRAN subprogram.

8.12 VIRTUAL STATEMENT

A VIRTUAL statement is used to specify the symbolic names and number of dimensions of virtual arrays to be used in IDENTIFY statements.

The form of a VIRTUAL statement is:

```
VIRTUAL a(d)[,a(d)]...
```

where each a(d) is an allocatable array declarator (5.2.2.3).

8.0 SPECIFICATION STATEMENTS
8.12 VIRTUAL STATEMENT

Each symbolic name *a* appearing in a VIRTUAL statement declares *a* to be a virtual array in that program unit, and defines the rank of *a* throughout the program unit.

Only one appearance of a symbolic name as an array name in an array declarator in a program unit is permitted.

9.0 DATA STATEMENT

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

All initially defined entities are defined when an executable program begins execution. All entities not initially defined, nor associated with an initially defined entity, are undefined at the beginning of execution of an executable program.

9.1 FORM OF A DATA STATEMENT

The form of a DATA statement is:

DATA _nlist_ /_clist_/ [[,] _nlist_ /_clist_/]...

where: _nlist_ is a list (2.11) of variable names, array names, array element names, substring names, and implied-DO lists

clist is a list of the form:

_a_ [,_a_]...

where _a_ is one of the forms:

_c_
_r_*_c_
_r_(_c_[,_c_]...)

_c_ is a constant or the symbolic name of a constant

_r_ is a nonzero, unsigned, integer constant or the symbolic name of such a constant. The _r_*_c_ form is equivalent to _r_ successive appearances of the constant _c_. The _r_(_c_[,_c_]...) form is equivalent to _r_ successive appearances of the list _c_[,_c_]... in _nlist_.

9.2 DATA STATEMENT RESTRICTIONS

Names of dummy arguments, functions, virtual arrays, and entities in blank common (including entities associated with an entity in blank common) must not appear in the list _nlist_.

There must be the same number of items specified by each list _nlist_ and its corresponding list _clist_. There is a one-to-one correspondence between the items specified by _nlist_ 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. If 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 determined by the array element subscript value (5.3.4).

The type of the nlist entity and the type of the corresponding clist constant must agree when either is of type character, bit, or logical. When the nlist entity is of type integer, real, double precision, half precision, Boolean, or complex, corresponding clist constant must also be of type integer, real, double precision, half precision, Boolean, or complex; if necessary, the clist constant is converted to the type of the nlist entity according to the rules for arithmetic conversion (Table 4), or Boolean conversion (10.6). Note that, if an nlist entity is of type double precision and the clist constant is of type real, the processor may supply more precision derived from the constant than can be contained in a real datum. If an nlist entity is of type double precision and the clist constant is of type half precision, the processor may supply more precision derived from the constant than can be contained in a half precision datum. If an nlist entity is of type real and the clist constant is of type half precision, the processor may supply more precision derived from the constant than can be contained in a half precision datum.

Any variable, array element, or substring may be initially defined except for:

(1) an entity that is a dummy argument,

(2) an entity in blank common, which includes an entity associated with an entity in blank common, or

(3) a variable in a function subprogram whose name is also the name of the function subprogram or an entry in the function subprogram, or

(4) an entity that is a virtual array.

A variable, array element, or substring must not be initially defined more than once in an executable program. If two entities are associated, only one may be initially defined in a DATA statement in the same executable program. Note that zero sized arrays and zero length substrings are initially defined and therefore must not be defined in a DATA statement.

Each subscript expression in the list nlist must be an extended integer constant expression except for implied-DO-variables as noted in 9.3. Each substring expression in the list nlist must be an

extended integer constant expression.

9.3 IMPLIED-DO IN A DATA STATEMENT

The form of an implied-DO list in a DATA statement is:

(_dlist_, _i_ = _m1_, _m2_ [,_m3_])

where: _dlist_ is a list of array element names and implied-DO lists

_i_ is the name of an integer variable, called the _implied-DO-variable_

_m1_, _m2_, _m3_ are each a scalar extended integer constant expression, except that the expression may contain implied-DO-variables of other implied-DO lists that have this implied-DO list within their ranges.

The range of an implied-DO list is the list _dlist_. An iteration count and the values of the implied-DO-variable are established from _m1_, _m2_, and _m3_ exactly as for a DO-loop (11.10), except that the iteration count must be positive. When an implied-DO list appears in a DATA statement, the list items in _dlist_ are specified once for each iteration of the implied-DO list with the appropriate substitution of values for any occurrence of the implied-DO-variable _i_. The appearance of an implied-DO-variable name in a DATA statement does not affect the definition status of a variable of the same name in the same program unit.

Each subscript expression in the list _dlist_ must be an extended integer constant expression, except that the expression may contain implied-DO-variables of implied-DO lists that have the subscript expression within their ranges.

The following is an example of a DATA statement that contains implied-DO lists:

```
DATA (( X(J,I), I=1,J), J=1,5) / 15*0.  /
```

9.4 CHARACTER CONSTANT IN A DATA STATEMENT

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 initially defined 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 initial definition of a character entity causes definition of all of the characters in the entity, and that each character constant initially defines exactly one variable, array element, or substring.

10.0 ASSIGNMENT STATEMENTS

Completion of execution of an assignment statement causes definition of an entity, except in the case of the IDENTIFY statement.

There are seven kinds of assignment statements:

(1) Arithmetic

(2) Logical

(3) Statement label (ASSIGN)

(4) Character

(5) Boolean

(6) Bit

(7) Array assignment statements (IDENTIFY and FORALL)

10.1 ARITHMETIC ASSIGNMENT STATEMENT

The form of an arithmetic assignment statement is:

$\underline{v} = \underline{e}$

where: $\underline{v}$ is the name of a variable, array, array section, or array element of type integer, real, double precision, half precision, or complex

$\underline{e}$ is an arithmetic expression or a Boolean expression

Execution of an arithmetic assignment statement causes the evaluation of the expression $\underline{e}$ by the rules in Section 6, conversion of $\underline{e}$ to the type of $\underline{v}$, and definition and assignment of $\underline{v}$ with the resulting value, as established by the rules in Table 4. If $\underline{v}$ is an array name or an array section name, the statement is an arithmetic array assignment statement and $\underline{e}$ must be conformable with $\underline{v}$. If $\underline{v}$ is a scalar, $\underline{e}$ must be a scalar. If $\underline{e}$ is a scalar and $\underline{v}$ is an array name or array section name, the scalar value is treated as if it had been extended to an array with the shape of $\underline{v}$ in which all elements have the value $\underline{e}$ and then assigned.

Table 4

Arithmetic Conversion and Assignment of e to v

Type of v	Value Assigned
Integer	INT(e)
Real	REAL(e)
Double Precision	DBLE(e)
Half Precision	HALF(e)
Complex	CMPLX(e)

The functions in the "Value Assigned" column of Table 4 are generic functions described in Table 5 (16.10).

10.2 LOGICAL ASSIGNMENT STATEMENT

The form of a logical assignment statement is:

v = e

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

e is a logical or bit expression

Execution of a logical assignment statement causes the evaluation of the logical expression e followed by the assignment and definition of v with the value of e. Note that e must have a value of type logical or bit. If v is an array name or an array section name, the statement is a logical array assignment statement and e must be conformable with v. If v is a scalar, e must be a scalar. If e is a scalar and v is an array name or array section name, the scalar value is treated as if it had been extended to an array with the shape of v in which all elements have the value e and then assigned. If e is of type bit, e is converted to type logical (BTOL(e)) and then assigned.

10.3 STATEMENT LABEL ASSIGNMENT (ASSIGN) STATEMENT

The form of a statement label assignment statement is:

ASSIGN s TO i

where: s is a statement label

i is an integer variable name

Execution of an ASSIGN statement causes the statement label s to be assigned t the integer variable i. The statement label must be the label of a statement that appears in the same program unit as the ASSIGN statement. The statement label must be the label of an executable statement or a FORMAT statement.

Execution of a statement label assignment statement is the only way that a variable may be defined with a statement label value.

A variable must be defined with a statement label value when referenced in an assigned GO TO statement (11.3) or as a format identifier (13.4) in an input/output statement. While defined with a statement label value, the variable must not be referenced in any other way.

An integer variable defined with a statement label value may be redefined with the same or a different statement label value or an integer value.

10.4 CHARACTER ASSIGNMENT STATEMENT

The form of a character assignment statement is:

v = e

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

e is a character expression

Execution of a character assignment statement causes the evaluation of the expression e followed by the assignment and definition of v with the value of e. None of the character positions being defined in v may be referenced in e. v and e may have different lengths. If the length of v is greater than the length of e, the effect is as though e were 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 were truncated from the right until it is the same length as v and then assigned. If v is an array name or an array section name, the statement is a *character array assignment statement* and e must be conformable with v. If v is a scalar, e must be a scalar. If e is a scalar and v is an array name or array section name, the scalar value is treated as if it had been extended to an array with the shape of v in which all elements have the value e and then assigned.

Only as much of the value of e must be defined as is needed to define v. In the example:

```
CHARACTER A*2, B*4
A=B
```

the assignment A=B requires that the substring B(1:2) be defined. It does not require that the substring B(3:4) be defined.

If v is a substring, e is assigned only to the substring. The definition status of substrings not specified by v is unchanged.

10.5 MULTIPLE ASSIGNMENT STATEMENT

The form of a multiple assignment statement is:

v = [v=]...e

where:

- v is the name of a variable, array, array section, array element, or character substring
- e is an expression. The types of the elements v and the expression e must be such that the form:

  v = e

is a valid assignment statement for each v in the multiple assignment statement.

Execution of a multiple assignment statement causes the evaluation of the expression e. After any necessary conversion, the assignment and definition of the rightmost v with value of e occurs. Assignment and definition of each additional v occurs in right-to-left order. The value assigned to each v is the value of the v immediately to its right, after any necessary conversion.

10.6 BOOLEAN ASSIGNMENT STATEMENT

The form of a Boolean assignment statement is:

v = e

where:

- v is the name of a Boolean variable, Boolean array, Boolean array section, or Boolean array element
- e is a Boolean expression or an arithmetic expression

Execution of a Boolean assignment statement causes the evaluation of the expression e by the rules in Section 6, and the conversion, assignment, and definition of v.

If e is an arithmetic expression, the value assigned to v is BOOL(e).

If v is an array name or an array section name, the statement is a Boolean array assignment statement and e must be conformable with v. If v is a scalar, e must be a scalar. If e is a scalar and v is an array name or array section name, the scalar value is treated as if it had been extended to an array with the shape of v in which all elements have the value e and then assigned.

10.7 BIT_ASSIGNMENT_STATEMENT

The form of a bit assignment statement is:

    v=e

where: v is the name of a bit variable, bit array, bit array section, or bit array element

e is a bit expression or a logical expression

Execution of a bit assignment statement causes the evaluation of the bit expression e followed by the assignment and definition of v with the value of e. Note that e must have a value of type bit or logical. If v is an array name or an array section name, the statement is a bit array assignment statement and e must be conformable with v. If v is a scalar, e must be a scalar. If e is a scalar and v is an array name or array section name, the scalar value is treated as if it had been extended to an array with the shape of v in which all elements have the value e and then assigned. If e is of type logical, e is converted to type bit (LTOB(e)) and then assigned to v.

10.8 ARRAY_ASSIGNMENT_STATEMENTS

10.8.1 IDENTIFY_STATEMENT

An IDENTIFY statement allows specification of sections of arrays which may consist of logically noncontiguous elements of a parent array, or which may be skewed with respect to the orthogonal dimensions (axes) of a parent array.

The form of an IDENTIFY statement is:

    IDENTIFY (rv[,rv]...)v(i[,i]...)=r(m[,m]...)

where:

- v is the identified or "virtual" array name
- r is the host or "parent" array
- i is an unsubscripted integer variable name to be used as a dummy subscript variable in none or more of the linear mapping expressions defining v. The set of dummy subscript variables specified in the list (i[,i]...) taken in order of appearance in the list define the ordinal order of the mapping of v onto a subset of r. The number of dummy subscript variables in the list defines the rank of v.
- m is a scalar-valued integer expression that is linear in the dummy subscript variables. The set of m specified in the list (m[,m]...) defines the mapping of v onto r in terms of the dummy subscript variables. The number of expressions in the list (m,[,m]...) must equal the rank of the host or parent array r.
- rv is a range declarator. The number of range declarators in the list, (rv[,rv]...) must equal the number of dummy subscript variables in the list (i[,i]...), i.e. it must equal the rank of v.

The range declarator is of the form:

```
[r1:]r2
```

where:

- r1 is a lower range bound specification
- r2 is an upper range bound specification

If r1 is omitted, it defaults to the value one. A lower or upper range bound must be a scalar-valued integer expression.

The identified array name v must have been previously declared as an array name in a VIRTUAL statement only. v and r must not be the same name.

The identified array name v may appear undimensioned in a type specification statement, but must not appear declared as an array name in any specification statement, other than a VIRTUAL statement. The identified array must be of the same type as the host array.

An identified array must not be referenced or defined until after it

has been "identified" by execution of an IDENTIFY statement, except when referenced in a NAMELIST I/O list.

The number of entries in the list of dummy subscript variables (i[,i]...) for v must equal the initial declared rank of v.

The dummy subscript variables (i[,i]...) are utilized only in a formal sense; values assigned to the variables (names) elsewhere in the program unit are unaltered by the occurrence of these variables (names) in the IDENTIFY statement.

The scalar-valued subscript quantities (m[,m]...) must be mathematically equivalent to expressions of the form k0+kxi1+k2xi2+...+knxin in which the ks are scalar integer expressions, not involving the dummy subscript variables, and the is are the dummy subscript variables used to define the mapping of the virtual array to the host array.

The host array r may be an explicitly declared "real" array or a previously identified "virtual" array.

The host array r may be an allocated allocatable array. If r subsequently appears in a FREE statement, then v becomes undefined.

The host array r may not be an assumed-size array, a dummy argument array, or an array-valued function name in the function.

The array r may have at most rank seven; similarly, the rank of v may be at most seven.

The scalar-valued subscript quantities (m[,m]...) along with the dummy subscript variables (i[,i]...) specify the selection mapping of the identified array. At the time of execution of an IDENTIFY statement, the scalar-valued subscript quantities (m[,m]...) define the mapping of v onto r. Variables involved in m may be redefined or become indefined during execution of subsequent executable statments with no effect on the mapping specified by the IDENTIFY statement. The order in which the "dummy subscript variables" appear in the list (i[,i]...) defines the specific ordering in which the virtual array v is to be defined and referenced. If a "dummy subscript variable" included in the list (i[,i]...) is not utilized in the list of "subscript quantities" (m[,m]...), in general, a replication of the real array r is implied. If the mapping of the virtual array is such that two or more virtual array elements map to a single array element in the host array, the virtual array may be referenced but not defined or redefined.

When an identified array or section of an identified array is passed as an argument to a subprogram, only elements within the identified "virtual" array or array sction are passed to the subprogram.

Once defined by an IDENTIFY statement, identified arrays may be utilized in exactly the same manner as any declared "real" array. In particular, an identified array may be subscripted or sectioned, and an identified array or section of an identified array may be passed as an argument to a subprogram in the same manner as a declared "real" array.

10.8.2 FORALL STATEMENT

The FORALL statement is used to represent an array assignment in terms of indices. The form of the statement is:

FORALL(*index*=*range*[,*index*=*range*]...[,*lexp*])*st*

where:

index is an integer variable

range has the form *ie*1:*ie*2[:*ie*3] where *ie*1, *ie*2, and *ie*3 are integer expressions that do not involve any indices of the statement; if *ie*1 is absent it is as if it were present with the value 1. *ie*3 must not be zero.

lexp is a scalar logical expression, which may involve the indices. The effect is as if *lexp* is evaluated for all index combinations prior to the execution of the statement *st*.

st is an assignment statement of the form:

aes=*exp*

where:

aes is an array element or an array section. It must reference all the indices

exp is an expression

An index can take only the values:

*m*1+*k*x*m*3,*k*=1,2,...,INT((*m*2-*m*1+*m*3)/*m*3)

where *m*1, *m*2, and *m*3 are the values of *ie*1, *ie*2, and *ie*3 on entry to the statement. An index for which INT((*m*2-*m*1+*m*3)/*m*3) ≤0 cannot take any values. Execution of the FORALL statement consists of the evaluation in any order of *exp* for all valid combinations of index values for which *lexp* is true, followed by the assignment of these values to the corresponding entities *aes*. If *lexp* is omitted, it is as if it were present with value .TRUE. The statement must not

cause any entity referenced as all or part of <u>aes</u> to be assigned a value more than once. The scope of the indices is the FORALL statement itself and any uses of their names outside the statement are references to separate entities. The indices must not be altered by a function referenced during the evaluation of <u>exp</u>.

11.0 CONTROL STATEMENTS

Control statements may be used to control the execution sequence.

There are sixteen control statements:

(1) Unconditional GO TO

(2) Computed GO TO

(3) Assigned GO TO

(4) Arithmetic IF

(5) Logical IF

(6) Block IF

(7) ELSE IF

(8) ELSE

(9) END IF

(10) DO

(11) CONTINUE

(12) STOP

(13) PAUSE

(14) END

(15) CALL

(16) RETURN

(17) Logical WHERE

(18) Block WHERE

(19) OTHERWISE

(20) ENDWHERE

The CALL and RETURN statements are described in Section 16.

11.1 UNCONDITIONAL GO TO STATEMENT

The form of an unconditional GO TO statement is:

GO TO s

where s is the statement label of an executable statement that appears in the same program unit as the unconditional GO TO statement.

Execution of an unconditional GO TO statement causes a transfer of control so that the statement identified by the statement label is executed next.

11.2 COMPUTED GO TO STATEMENT

The form of a computed GO TO statement is:

GO TO (s [,s]...) [,] e

where: e is an arithmetic or Boolean scalar expression

s is the statement label of an executable statement that appears in the same program unit as the computed GO TO statement. The same statement label may appear more than once in the same computed GO TO statement.

Execution of a computed GO TO statement causes evaluation of the expression e. The value obtained is converted to type integer, if necessary, by application of the intrinsic function INT to yield an integer value i.

11.3 ASSIGNED GO TO STATEMENT

The form of an assigned GO TO statement is:

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

where: i is an integer variable name

s is the statement label of an executable statement that appears in the same program unit as the assigned GO TO statement. The same statement label may appear more than once in the same assigned GO TO statement.

At the time of execution of an assigned GO TO statement, the variable i must be defined with the value of a statement label of an executable statement that appears in the same program unit. Note that the variable may be defined with a statement label value only by an ASSIGN statement (10.3) in the same program unit as the assigned GO TO statement. The execution of the assigned GO TO

statement causes a transfer of control so that the statement identified by that statement label is executed next.

If the parenthesized list is present, the statement label assigned to i must be one of the statement labels in the list.

11.4 ARITHMETIC IF STATEMENT

The form of an arithmetic IF statement is:

IF (e) s1 , s2 , s3

where: e is an integer, real, double precision, half precision, or Boolean scalar expression

s1, s2, and s3 are each the statement label of an executable statement that appears in the same program unit as the arithmetic IF statement. The same statement label may appear more than once in the same arithmetic IF statement.

Execution of an arithmetic IF statement causes evaluation of the expression e followed by a transfer of control. The statement identified by s1, s2, or s3 is executed next as the value of e is less than zero, equal to zero, or greater than zero, respectively. If e is a Boolean expression, the value of INT(e) is used.

11.5 LOGICAL IF STATEMENT

The form of a logical IF statement is:

IF (e) st

where: e is a scalar logical expression

st is any executable statement except a DO, block IF, ELSE IF, ELSE, END IF, END, block WHERE, OTHERWISE, END WHERE, logical WHERE, or another logical IF statement.

Execution of a logical IF statement causes evaluation of the expression e. If the value of e is true, statement st is executed. If the value of e is false, statement st is not executed and the execution sequence continues as though a CONTINUE statement were executed.

Note that the execution of a function reference in the expression e of a logical IF statement is permitted to affect entities in the statement st.

11.6 BLOCK IF STATEMENT

The block IF statement is used with the END IF statement and, optionally, the ELSE IF and ELSE statements to control the execution sequence.

The form of a block IF statement is:

IF (e) THEN

where e is a scalar logical expression.

11.6.1 IF-LEVEL.

The IF-level of a statement s is

n1 - n2

where n1 is the number of block IF statements from the beginning of the program unit up to and including s, and n2 is the number of END IF statements in the program unit up to but not including s.

The IF-level of every statement must be zero or positive. The IF-level of each block IF, ELSE IF, ELSE, and END IF statement must be positive. The IF-level of the END statement of each program unit must be zero.

11.6.2 IF-BLOCK.

An IF-block consists of all of the executable statements that appear following the block IF statement up to, but not including, the next ELSE IF, ELSE, or END IF statement that has the same IF-level as the block IF statement. An IF-block may be empty.

11.6.3 EXECUTION OF A BLOCK IF STATEMENT.

Execution of a block IF statement causes evaluation of the expression e. If the value of e is true, normal execution sequence continues with the first statement of the IF-block. If the value of e is true and the IF-block is empty, control is transferred to the next END IF statement that has the same IF-level as the block IF statement. If the value of e is false, control is transferred to the next ELSE IF, ELSE, or END IF statement that has the same IF-level as the block IF statement.

Transfer of control into an IF-block from outside the IF-block is prohibited.

If the execution of the last statement in the IF-block does not result in a transfer of control, control is transferred to the next END IF statement that has the same IF-level as the block IF

statement that precedes the IF-block.

11.7 ELSE IF STATEMENT

The form of an ELSE IF statement is:

ELSE IF (*e*) THEN

where *e* is a scalar logical expression.

11.7.1 ELSE IF-BLOCK.

An *ELSE IF-block* consists of all of the executable statements that appear following the ELSE IF statement up to, but not including, the next ELSE IF, ELSE, or END IF statement that has the same IF-level as the ELSE IF statement. An ELSE IF-block may be empty.

11.7.2 EXECUTION OF AN ELSE IF STATEMENT.

Execution of an ELSE IF statement causes evaluation of the expression *e*. If the value of *e* is true, normal execution sequence continues with the first statement of the ELSE IF-block. If the value of *e* is true and the ELSE IF-block is empty, control is transferred to the next END IF statement that has the same IF-level as the ELSE IF statement. If the value of *e* is false, control is transferred to the next ELSE IF, ELSE, or END IF statement that has the same IF-level as the ELSE IF statement.

Transfer of control into an ELSE IF-block from outside the ELSE IF-block is prohibited. The statement label, if any, of the ELSE IF statement must not be referenced by any statement.

If execution of the last statement in the ELSE IF-block does not result in a transfer of control, control is transferred to the next END IF statement that has the same IF-level as the ELSE IF statement that precedes the ELSE IF-block.

11.8 ELSE STATEMENT

The form of an ELSE statement is:

ELSE

11.8.1 ELSE-BLOCK.

An *ELSE block* consists of all of the executable statements that appear following the ELSE statement up to, but not including, the next END IF statement that has the same IF-level as the ELSE statement. An ELSE-block may be empty.

An END IF statement of the same IF-level as the ELSE statement must

appear before the appearance of an ELSE IF or ELSE statement of the same IF-level.

11.8.2 EXECUTION OF AN ELSE STATEMENT.

Execution of an ELSE statement has no effect.

Transfer of control into an ELSE-block from outside the ELSE-block is prohibited. The statement label, if any, of an ELSE statement must not be referenced by any statement.

11.9 END IF STATEMENT

The form of an END IF statement is:

```
END IF
```

Execution of an END IF statement has no effect.

For each block IF statement, there must be a corresponding END IF statement in the same program unit. A _corresponding END IF statement_ is the next END IF statement that has the same IF-level as the block IF statement.

11.10 DO STATEMENT

A DO statement is used to specify a loop, called a _DO-loop_.

The form of a DO statement is:

DO _s_ [,] _i_ = _e1_, _e2_ [,_e3_]

where: _s_ is the statement label of an executable statement. The statement identified by _s_, called the _terminal statement_ of the DO-loop, must follow the DO statement in the sequence of statements within the same program unit as the DO statement.

_i_ is the name of an integer, real, half precision, or double precision variable, called the _DO variable_

_e1_, _e2_, and _e3_ are each an integer, real, half precision, double precision, or Boolean scalar expression

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

11.10.1 RANGE OF A DO-LOOP.

The range of a DO-loop consists of all of the executable statements that appear following the DO statement that specifies the DO-loop, to and including the terminal statement of the DO-loop.

If a DO statement appears within the range of a DO-loop, the range of the DO-loop specified by that DO statement must be contained entirely within the range of the outer DO-loop. More than one DO-loop may have the same terminal statement.

If a DO statement appears within an IF-block, ELSE IF-block, or ELSE-block, the range of that DO-loop must be contained entirely within that IF-block, ELSE IF-block, or ELSE-block, respectively.

If a block IF statement appears within the range of a DO-loop, the corresponding END IF statement must also appear within the range of that DO-loop.

11.10.2 ACTIVE AND INACTIVE DO-LOOPS.

A DO-loop is either active or inactive. Initially inactive, a DO-loop becomes active only when its DO statement is executed.

Once active, the DO-loop becomes inactive only when:

(1) its iteration count is tested (11.10.4) and determined to be zero,

(2) its DO-variable becomes undefined or is redefined by means other than the incrementation described in 11.10.7,

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

(4) a RETURN, STOP, or END statement is executed in its program unit, or

(5) it is in the range of another DO-loop whose DO-statement is executed.

Execution of a function reference or CALL statement that appears in the range of a DO-loop does not cause the DO-loop to become inactive. Note that transfer of control out of the range of a DO-loop does not inactivate the DO-loop. However, the DO-loop becomes inactive if the DO-variable becomes undefined or is redefined outside the range.

When a DO-loop becomes inactive, the DO-variable of the DO-loop retains its last defined value, unless it has become undefined.

11.10.3 EXECUTING A DO STATEMENT.

The effect of executing a DO statement is to perform the following steps in sequence:

(1) The *initial parameter* m1, the *terminal parameter* m2, and the *incrementation parameter* m3 are established by evaluating e1, e2, and e3, respectively, including, if necessary, conversion to the type of the DO-variable according to the rules for arithmetic conversion (Table 4). If e3 does not appear, m3 has a value of one. m3 must not have a value of zero.

(2) The DO-variable becomes defined with the value of the initial parameter m1.

(3) The iteration count is established and is the value of the expression

MAX(INT((m2-m1+m3)/m3),mtc)

where: mtc is the minimum-trip-count parameter. mtc has a value of either one or zero, and is established at processor invocation. The value of mtc may be dynamically modified by the minimum-trip-count control directive (3.7.4).

Note that the iteration count is equal to the value of mtc whenever:

$m1 > m2$ and $m3 > 0$, or

$m1 < m2$ and $m3 < 0$.

At the completion of execution of the DO statement, loop control processing begins.

11.10.4 LOOP CONTROL PROCESSING.

Loop control processing determines if further execution of the range of the DO-loop is required. The iteration count is tested. If it is not zero, execution of the first statement in the range of the DO-loop begins. If the iteration count is zero, the DO-loop becomes inactive. If, as a result, all of the DO-loops sharing the terminal statement of this DO-loop are inactive, normal execution continues with execution of the next executable statement following the terminal statement. However, if some of the DO-loops sharing the terminal statement are active, execution continues with incrementation processing, as described in 11.10.7.

11.10.5 EXECUTION OF THE RANGE.

Statements in the range of a DO-loop are executed until the terminal statement is reached. Except by the incrementation described in 11.10.7, the DO-variable of the DO-loop may neither be redefined nor become undefined during execution of the range of the DO-loop.

11.10.6 TERMINAL STATEMENT EXECUTION.

Execution of the terminal statement occurs as a result of the normal execution sequence or as a result of transfer of control, subject to the restrictions in 11.10.8. Unless execution of the terminal statement results in a transfer of control, execution then continues with incrementation processing, as described in 11.10.7.

11.10.7 INCREMENTATION PROCESSING.

Incrementation processing has the effect of the following steps performed in sequence:

(1) The DO-variable, the iteration count, and the incrementation parameter of the active DO-loop whose DO statement was most recently executed, are selected for processing.

(2) The value of the DO-variable is incremented by the value of the incrementation parameter m3.

(3) The iteration count is decremented by one.

(4) Execution continues with loop control processing (11.10.4) of the same DO-loop whose iteration count was decremented.

An example illustrates the above:

```
          N=0
          DO 100 I=1,10
          J=I
          DO 100 K=1,5
          L=K
      100 N=N+1
      101 CONTINUE
```

After execution of the above statements and at the execution of the CONTINUE statement, I=11, J=10, K=6, L=5, and N=50. Also consider the following example (mtc=0):

```
          N=0
          DO 200 I=1,10
          J=I
          DO 200 K=5,1
          L=K
```

```
       200 N=N+1
       201 CONTINUE
```

After execution of the above statements and at the execution of the CONTINUE statement, I=11, J=10, K=5, and N=0. L is not defined by the above statements.

11.10.8 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 to any executable statement in the range of an active DO-loop is permitted unless the statement is also in the range of an inactive DO-loop or the transfer of control is not permitted by the rules for execution of an IF-block, ELSE IF-block, ELSE-block, WHERE-block, or OTHERWISE-block.

11.11 CONTINUE_STATEMENT

The form of a CONTINUE statement is:

```
          CONTINUE
```

Execution of a CONTINUE statement has no effect.

If the CONTINUE statement is the terminal statement of a DO-loop, the next statement executed depends on the result of the DO-loop incrementation processing (11.10.7).

11.12 STOP_STATEMENT

The form of a STOP statement is:

```
          STOP [n]
```

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

Execution of a STOP statement causes termination of execution of the executable program. At the time of termination, the digit string or character constant is accessible.

11.13 PAUSE_STATEMENT

The form of a PAUSE statement is:

```
          PAUSE [n]
```

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

Execution of a PAUSE statement causes a cessation of execution of

the executable program. Execution is resumable. At the time of cessation of execution, the digit string or character constant is accessible via display to the operator, and also to the user if the latter is in communication with the processor during execution. Resumption of execution is not under control of the program. If execution is resumed, the execution sequence continues as though a CONTINUE statement were executed. See the reference manual for the appropriate operating system regarding the means for indicating that execution should be resumed. If the program is executing interactively, the user causes execution to resume by entering an input line consisting of the characters GO in columns one and two. The alphabetic case of the letters in GO is not significant.

11.14 END STATEMENT

The END statement indicates the end of the sequence of statements and comment lines of a program unit (3.5). If executed in a function or subroutine subprogram, it has the effect of a RETURN statement (16.8). If executed in a main program, it terminates the execution of the executable program.

The form of an END statement is:

```
END
```

An END statement is written only in columns 7 through 72 of an initial line. An END statement must not be continued. No other statement in a program unit may have an initial line that appears to be an END statement.

The last line of every program unit must be an END statement.

11.15 LOGICAL WHERE STATEMENT

The logical WHERE statement is used to control the assignment of values in an array assignment, and to control the evaluation of expressions in the array assignment statement, according to the value of a logical array expression.

The form of a logical WHERE statement is:

WHERE (*lae*) *st*

where:

lae is an array expression of type logical or bit

st is an array assignment statement

Execution of a logical WHERE statement has the same effect as execution of the sequence

```
WHERE(lae)
st
END WHERE
```

11.16 BLOCK WHERE STATEMENT

The block WHERE statement is used with the END WHERE statement, and optionally the OTHERWISE statement to control the assignment of values in a block of array assignment statements, and to control the evaluation of expressions in array assignment statements, according to the value of a logical array expression.

The form of a block WHERE statement is:

```
WHERE (lae)
```

where: lae is an array expression of type logical or bit

11.16.1 WHERE-LEVEL

The WHERE-level of a statement s is

n1 - n2

where n1 is the number of block WHERE statements from the beginning of the program unit up to and including s, and n2 is the number of END WHERE statements in the program unit up to but not including s.

The WHERE-level of every statement must be zero or one. Note that this precludes nesting BLOCK WHERE statements. The WHERE-level of each block WHERE, OTHERWISE, and END WHERE statement must be one. The WHERE-level of the END statement of each program unit must be zero.

11.16.2 WHERE-BLOCK

A WHERE-block consists of all of the executable statements that appear following the block WHERE statement up to, but not including, the next OTHERWISE or END WHERE statement that has the same WHERE-level as the block WHERE statement. All of the executable statements in a WHERE-block must be array assignment statements that do not involve an array-valued function reference. In each array assignment statement of the form v=[v=]...a, v must have the same shape as lae. No statement in a WHERE-block can be the terminal statement of a DO. A WHERE-block may be empty. Transfer of control into a WHERE-block is prohibited.

11.16.3 EXECUTION OF A BLOCK WHERE STATEMENT

Execution of a block WHERE statement causes evaluation of the expression lae. The statements in the WHERE-block are executed in

normal execution sequence. When an array assignment statement is executed under control of a block WHERE statement, the semantics of the statement are as if it were executed as follows:

(1) The evaluation of the expression _e_ takes place as if it were evaluated for all elements where _lae_ is true and the result of the evaluation assigned to the corresponding element of _v_. At points where _lae_ is false, the value of the corresponding element of _v_ is not altered and it is as if the expression _e_ was not evaluated.

(2) Assignment statements between a WHERE and an ENDWHERE (including an optional OTHERWISE block) are executed in normal execution sequence. Values defined in _v_ in one assignment statement may be referenced or defined in subsequent assignment statements.

The value of _lae_ is not affected by the execution of statements in the WHERE-block.

Transfer of control into a WHERE-block is prohibited.

11.17 OTHERWISE STATEMENT

The form of an OTHERWISE statement is:

    OTHERWISE

11.17.1 OTHERWISE-BLOCK

An _OTHERWISE-BLOCK_ consists of all of the executable statements that appear following the OTHERWISE statement up to, but not including, the END WHERE statement that has the same WHERE-level as the OTHERWISE statement. All of the executable statements in an OTHERWISE-block must be array assignment statements that do not involve an array-valued function reference. In array assignment statements of the form _v_=[_v_=]...._e_, _v_ must have the same shape as _lae_. No statement in an OTHERWISE-block can be the terminal statement of a DO. Transfer of control into an OTHERWISE-block is prohibited. An OTHERWISE-block may be empty.

11.17.2 EXECUTION OF AN OTHERWISE STATEMENT

Execution of an OTHERWISE statement causes the statements in the OTHERWISE-block to be executed in normal execution sequence. When an array assignment statement is executed in an OTHERWISE block, the semantics of the statement are as if it were executed as follows:

(1) The evaluation of the expression _e_ takes place as if it were evaluated for all elements where _lae_ is false and the result

of the evaluation assigned to the corresponding element of _v_. At points where _lae_ is true, the value of the corresponding element of _v_ is not altered and it is as if the expression _e_ was not evaluated.

(2) Assignment statements between a WHERE and an ENDWHERE (including an optional OTHERWISE block) are executed in normal execution sequence. Values defined in _v_ in one assignment statement may be referenced or defined in subsequent assignment statements.

The value of _lae_ is not affected by the execution of statements in the OTHERWISE-block.

Transfer of control into an OTHERWISE-block is prohibited.

11.18 END WHERE STATEMENT

The form of an END WHERE statement is:

```
END WHERE
```

Execution of an END WHERE statement has no effect. For each block WHERE statement, there must be a corresponding END WHERE statement in the same program unit. A _corresponding END WHERE statement_ is the next END WHERE statement that has the same WHERE-level as the block WHERE statement.

12.0 ARRAY STORAGE ALLOCATION

There are two executable statements that control allocation of storage for arrays:

(1) ALLOCATE statement

(2) FREE statement

12.1 ALLOCATE_STATEMENT

The ALLOCATE statement is used to allocate storage for an allocatable array. The form of an ALLOCATE statement is:

```
ALLOCATE ad[,ad]...
```

where: ad has the form of a constant or adjustable array declarator.

The array names appearing in ad must have been specified to be allocatable arrays. The dimension bound expressions are subject to the same limitations as in an array declarator, except that array element references are permitted to appear. No allocatable array name in an ALLOCATE statement may be referenced in a primary of any expression within the same ALLOCATE statement. For example,

```
ALLOCATE A(10,EXTENT(A,1)),B(SIZE(A))
```

is prohibited. The dimension bound expressions in ad are evaluated at the time of execution of the ALLOCATE statement. The values of the dimension bound expressions determine the sizes of the corresponding dimensions for the allocatable array and the upper and lower bounds of the dimensions. The number of dimensions is determined by the allocatable array declarator and must correspond with the number specified in ad. If the allocatable array was declared with an explicit lower bound dl, then the corresponding lower bound of the array declarator ad must specify the same value An allocatable array must not be defined or referenced if it is not currently allocated.

After execution of an ALLOCATE statement for an array, the properties of dimension size, lower and upper dimension bound, and array size are established. However, any entities in the dimension bounds expressions may be redefined or become undefined with no effect on the above mentioned properties. Immediately after an array is allocated the values of all of the array elements are undefined.

An ALLOCATE statement may appear in a main program, subroutine, or

function subprogram. At the beginning of execution of an executable program, all allocatable arrays have a state of "not currently allocated". All allocatable arrays that have been allocated storage by an ALLOCATE and that have not been subsequently freed, will be freed automatically if execution of the subprogram in which they were allocated is terminated by execution of a RETURN or END statement, and the allocatable array name is not specified in a SAVE statement (explicitly or implicitly). Allocating a currently allocated array is prohibited.

12.2 FREE STATEMENT

The FREE statement causes the storage for previously allocated arrays to be released. The form of a FREE statement is:

```
FREE a[,a]...
```

where: a is the name of an allocatable array previously allocated by execution of an ALLOCATE statement. Results are undefined if a is not currently allocated.

13.0 INPUT/OUTPUT STATEMENTS

Input statements provide the means of transferring data from external media to internal storage or from an internal file to internal storage. This process is called _reading_. Output statements provide the means of transferring data from internal storage to external media or from internal storage to an internal file. This process is called _writing_. Some input/output statements specify that editing of the data is to be performed.

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 connection to the external medium.

There are fourteen input/output statements:

(1) READ

(2) WRITE

(3) PRINT

(4) OPEN

(5) CLOSE

(6) INQUIRE

(7) BACKSPACE

(8) ENDFILE

(9) REWIND

(10) PUNCH

(11) BUFFER IN

(12) BUFFER OUT

(13) ENCODE

(14) DECODE

The READ, WRITE, PRINT, PUNCH, BUFFER IN, BUFFER OUT, ENCODE, and DECODE statements are _data transfer input/output statements_ (13.8). The OPEN, CLOSE, INQUIRE, BACKSPACE, ENDFILE, and REWIND statements are _auxiliary input/output statements_ (13.10). The BACKSPACE, ENDFILE, and REWIND statements are _file positioning input/output_

statements (13.10.4).

13.1 RECORDS

A record is a sequence (2.1) of values or a sequence of characters. 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 three kinds of records:

(1) Formatted

(2) Unformatted

(3) Endfile

13.1.1 FORMATTED RECORD.

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 and depends primarily on the number of characters put into the record when it is written. However, it may depend on the processor and the external medium. The length may be zero. Formatted records may be read or written only by formatted input/output statements (13.8.1).

Formatted records may be prepared by some means other than FORTRAN; for example, some manual input device.

13.1.2 UNFORMATTED RECORD

An unformatted record consists of a sequence of values in a processor-dependent form and may contain both character and noncharacter data or may contain no data. The length of an unformatted record is measured in processor-dependent units and depends on the output list (13.8.2) used when it is written, as well as on the processor and the external medium. The length may be zero.

The only statements that read and write unformatted records are unformatted input/output statements (13.8.1), BUFFER IN, and BUFFER OUT statements (13.15). An unformatted record to be read by a BUFFER IN statement or to be written by a BUFFER OUT statement may not contain character, bit or half precision data.

13.1.3 ENDFILE RECORD.

An endfile record is written by an ENDFILE statement. An endfile record may occur only as the last record of a file. An endfile record does not have a length property.

13.2 FILES

A *file* is a sequence (2.1) of records.

There are two kinds of files:

(1) External

(2) Internal

13.2.1 FILE EXISTENCE.

At any given time, there is a processor-determined set of files that are said to *exist* for an executable program. A file may be known to the processor, yet not exist for an executable program at a particular time. For example, security reasons may prevent a file from existing for an executable program. A file may exist and contain no records; an example is a newly created file not yet written.

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.

All input/output statements may refer to files that exist. The INQUIRE, OPEN, CLOSE, WRITE, PRINT, PUNCH, BUFFER OUT, and ENDFILE statements may also refer to files that do not exist.

13.2.2 FILE PROPERTIES

At any given time, there is a processor-determined *set of allowed access methods*, a processor-determined *set of allowed forms*, and a processor-determined *set of allowed record lengths* for a file.

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 consisting of one to thirty-one alphabetic characters or digits. The first character of a file name must not be a digit. (An alpha character is a letter, upper or lower case, or one of the characters $, , # or _ [underline]. Upper and lower case letters in the file name are interpreted as variants of the same letter).

13.2.3 FILE POSITION.

A file that is connected to a unit (13.3) has a position property. Execution of certain input/output statements affects the position of a file. Certain circumstances can cause the position of a file to become indeterminate.

The *initial point* of a file is the position just before the first record. The *terminal point* is the position just after the last

record.

If a file is positioned within a record, that record is the *current record*; otherwise, there is no current record.

Let *n* be the number of records in the file. If 1< *i* ≤ *n* and a file is positioned within the *i*th record or between the (*i*-1)th record and the *i*th record, the (*i*-1)th record is the *preceding record*. If n ≥ 1 and a file is positioned at its terminal point, the preceding record is the *n*th and last record. If *n*=0 or if a file is positioned at its initial point or within the first record, there is no preceding record.

If 1 ≤ *i* <*n* and a file is positioned within the *i*th record or between the *i*th and (*i*+1)th record, the (*i*+1)th record is the *next record*. If n ≥ 1 and the file is positioned at its initial point, the first record is the next record. If *n*=0 or if a file is positioned at its terminal point or within the *n*th and last record, there is no next record.

13.2.4 FILE ACCESS.

There are two methods of accessing the records of an external file: sequential and direct. Some files may have more than one allowed access method; other files may be restricted to one access method. For example, a processor may allow only sequential access to a file on magnetic tape. Thus, the set of allowed access methods depends on the file and the processor.

The method of accessing the file is determined when the file is connected to a unit (13.3.2).

An internal file must be accessed sequentially.

13.2.4.1 Sequential Access.

When connected for sequential access, a file has the following properties:

(1) The order of the records is the order in which they were written if the direct access method is not a member of the set of allowed access methods for the file. If the direct access method is also a member of the set of allowed access methods for the file, the order of the records is the same as that specified for direct access (13.2.4.2). The first record accessed by sequential access is the record whose record number is 1 for direct access. The second record accessed by sequential access is the record whose record number is 2 for direct access, etc. A record that has not been written since the file was created must not be read.

(2) The records of the file are either all formatted or all unformatted, except that the last record of the file may be an endfile record.

(3) The records of the file must not be read or written by direct access input/output statements (13.8.1).

13.2.4.2 Direct Access.

When connected for direct access, a file has the following properties:

(1) The order of the records is the order of their record numbers. The records may be read or written in any order.

(2) The records of the file are either all formatted or all unformatted. If the sequential access method is also a member of the set of allowed access methods for the file, its endfile record, if any, is not considered to be part of the file while it is connected for direct access. If the sequential access method is not a member of the set of allowed access methods for the file, the file must not contain an endfile record.

(3) Reading and writing records is accomplished only by direct access input/output statements (13.8.1).

(4) All records of the file have the same length.

(5) Each record of the file is uniquely identified by a positive integer called the record number. The record number of a record is specified when the record is written. Once established, the record number of a record can never be changed. Note that a record may not be deleted; however, a record may be rewritten.

(6) 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 (13.3.2) to a unit. For example, it is permissible to write record 3, even though records 1 and 2 have not been written. Any record may be read from the file while it is connected to a unit, provided that the record was written since the file was created.

(7) The records of the file must not be read or written using list-directed formatting.

13.2.5 INTERNAL FILES

Internal files provide a means of transferring and converting data from internal storage to internal storage.

There are two types of internal files, standard and extended. A *standard internal file* is a sequence of character storage units. An *extended internal file* is a sequence of numeric storage units. Each type of internal file requires different input/output statement forms.

The standard or extended property of an internal file is established by the type of storage provided for the file.

Throughout this document, the phrase *internal file* shall be interpreted to mean *standard internal file*, unless explicitly prefixed with *extended*.

13.2.5.1 Standard Internal File Properties.

A standard internal file has the following properties:

(1) The file is a character variable, character array element, character array, or character substring.

(2) A record of an internal file is a character variable, character array element, or character substring.

(3) If the file is a character variable, character array element, or character substring, it consists of a single record whose length is the same as the length of the variable, array element, or substring, respectively. If the file is a character array, it is treated as a sequence of character array elements. Each array element is a record of the file. The ordering of the records of the file is the same as the ordering of the array elements in the array (5.3.4). Every record of the file has the same length, which is the length of an array element in the array.

(4) The variable, array element, or substring that is the record of the internal file becomes defined by writing the record. If the number of characters written in a record is less than the length of the record, the remaining portion of the record is filled with blanks.

(5) A record may be read only if the variable, array element, or substring that is the record is defined.

(6) A variable, array element, or substring that is a record of an internal file may become defined (or undefined) by means other than an output statement. For example, the variable,

array element, or substring may become defined by a character assignment statement.

(7) An internal file is always positioned at the beginning of the first record prior to data transfer.

13.2.5.2 Standard Internal File Restrictions.

A standard internal file has the following restrictions:

(1) Reading and writing records is accomplished only by sequential access formatted input/output statements (13.8.1) that do not specify list-directed formatting, or NAMELIST formatting (13.4 (6)).

(2) An auxiliary input/output statement must not specify an internal file.

13.2.5.3 Extended Internal File Properties

An extended internal file has the following properties

(1) The file is specified by a variable, array element, or array of type other than character, bit or half precision. If an array element is used, the file begins with the array element and extends through the end of the array. Otherwise, the file is the variable or array specified.

(2) A record of the file is one or more contiguous numeric storage units.

(3) The length of a record of the file is measured in characters, and is equal to

$a * m$

where:

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

m is the number of numeric storage units in the record

(4) Every record of the file has the same length.

(5) The variable or array element(s) that is a record of the file is defined by writing the record. If the number of characters written in a record is less than the length of the record, the remaining portion of the record is filled

with blanks.

(6) A record may be read only if the variable or array element(s) that is the record is defined.

(7) A variable or array element(s) that is a record of the file may become defined or undefined by means other than an output statement.

(8) An extended internal file is always positioned at the initial point of the first record prior to transfer.

13.2.5.4 Extended_Internal_File_Restrictions

An extended internal file has the following restrictions

(1) Reading and writing records is accomplished only by DECODE and ENCODE statements (13.14). List-directed or NAMELIST formatting must not be specified.

(2) An auxiliary input/output statement must not specify an extended internal files

13.3 UNITS

A unit is a means of referring to a file.

13.3.1 UNIT_EXISTENCE.

At any given time, there is a processor-determined set of units that are said to exist for an executable program. A unit exists for each allowed external unit identifier.

All input/output statements may refer to units that exist. The INQUIRE and CLOSE statements may also refer to units that do not exist.

13.3.2 CONNECTION_OF_A_UNIT.

A unit has a property of being connected or not connected. If connected, it refers to a file. A unit may become connected by preconnection, implicit connection, or by the execution of an OPEN statement. The property of connection is symmetric: if a unit is connected to a file, the file is connected to the unit.

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.

Implicit connection means that if a unit that is not connected is referenced in a data transfer or file positioning input/output statement it becomes connected to a processor-determined file immediately prior to the execution of the input/output statement. The name of the file to which the unit becomes connected is determined in the same manner as for an OPEN statement for which the FILE= specifier is omitted.

All input/output statements except OPEN, CLOSE, and INQUIRE must reference a unit that is connected to a file and thereby make use of or affect that file.

A file may be connected and not exist. An example is a preconnected new file.

The maximum number of units that may be connected to files at any one time is processor dependent, but must not be less than 50.

A unit must not be connected to more than one file at the same time, but an external file may be connected to more than one unit at the same time. However, means are provided to change the status of a unit and to connect a unit to a different file.

After a unit has been disconnected by the execution of a CLOSE statement, it may be connected again within the same executable program to the same file or a different file. After a file has been disconnected by the execution of a CLOSE statement, it may be connected again within the same executable program to the same unit or a different unit. Note, however, that the only means to refer to a file that has been disconnected is by its name in an OPEN or INQUIRE statement. Therefore, there may be no means to reconnect an unnamed file once it is disconnected.

13.3.3 UNIT SPECIFIER AND IDENTIFIER.

The form of a *unit specifier* is:

    [UNIT =] u

where *u* is an external unit identifier or an internal file identifier.

An external unit identifier is used to refer to an external file. An internal file identifier is used to refer to an internal file.

An *external unit identifier* is one of the following:

(1) A scalar integer or scalar Boolean expression *e* such that either

---

13.0 INPUT/OUTPUT STATEMENTS
13.3.3 UNIT SPECIFIER AND IDENTIFIER.

(a) INT(e) has an integer value in the range 0-999, or

(b) BOOL(e) has a value of the form L"f", where f is a sequence of one to seven letters or digits, starting with a letter, comprising a valid system file name. Upper and lower case letters in the file name are interpreted as variants of the same letter.

There is a correspondence between certain pairs of external units. In case (a), the unit identified by the value of INT(e) corresponds to the unit identified by "TAPEk", where k is the digit string, with no leading zeros, representing the value of INT(e). In case (b), if f is of the form TAPEk, where k is a digit string, with no leading zeros, representing an integer value in the range 0-999, then the unit identified by the value of BOOL(e) corresponds to the unit identified by the value of k; if f is not of this form, then no correspondence exists in this case.

A correspondence between two external units means that if one of the external units is connected to a file then the other unit is also connected to the file.

(2) An asterisk, identifying a particular processor-determined external unit that is preconnected for formatted sequential access (13.9.2).

The external unit identified by the value of e is the same external unit in all program units of the executable program. In the example:

```
SUBROUTINE A
READ (6) X
     .
     .
SUBROUTINE B
N=6
REWIND N
```

the value 6 used in both program units identifies the same external unit.

An external unit identifier in an auxiliary input/output statement (13.10) must not be an asterisk.

An internal file identifier provides the means of referring to a standard or extended internal file (13.2.5). An internal file identifier for a standard internal file is the symbolic name of a character variable, character array, character array section, character array element, or character substring. An internal file identifier for an extended internal file is the symbolic name of a

variable, array, array section, or array element of type other than character, bit or half-precision.

If the optional characters UNIT= are omitted from the unit specifier, the unit specifier must be the first item in a list of specifiers.

13.4 FORMAT SPECIFIER AND IDENTIFIER

The form of a *format specifier* is:

    [FMT =] f

where *f* is a format identifier.

A *format identifier* identifies a format. A format identifier must be one of the following:

(1) The statement label of a FORMAT statement that appears in the same program unit as the format identifier.

(2) An integer variable name that has been assigned the statement label of a FORMAT statement that appears in the same program unit as the format identifier (10.3).

(3) A character array name or character array section (14.1.2).

(4) Any scalar character expression except a character expression involving concatenation of a dummy argument or variable whose length specification is an asterisk in parentheses. Note that a character constant is permitted.

(5) An asterisk, specifying list-directed formatting.

(6) A NAMELIST group name.

(7) An array name or array section of type other than character, bit or half precision.

If the optional characters FMT= are omitted from the format specifier, the format specifier must be the second item in the control information list and the first item must be the unit specifier without the optional characters UNIT=.

13.5 RECORD SPECIFIER

The form of a *record specifier* is:

    REC = rn

where *rn* is an integer scalar expression whose value is positive.

It specifies the number of the record that is to be read or written in a file connected for direct access.

13.6 ERROR AND END-OF-FILE CONDITIONS

The set of input/output error conditions is processor dependent.

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

(1) An endfile record is encountered during the reading of a file conne In this case, the file is positioned after the endfile record.

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

If an error condition occurs during execution of an input/output statement, execution of the input/output statement terminates and the position of the file becomes indeterminate.

If an error condition or an end-of-file condition occurs during execution of a READ statement, execution of the READ statement terminates and the entities specified by the input list and implied-DO variables in the input list become undefined. Note that variables and array elements appearing only in subscripts, substring expressions, and implied-DO parameters in an input list do not become undefined when the entities specified by the list become undefined.

If an error condition occurs during execution of an output statement, execution of the output statement terminates and implied-DO-variables in the output list become undefined.

If an error condition occurs during execution of an input/output statement that contains neither an input/output status specifier (13.7) nor an error specifier (13.7.1), or if an end-of-file condition occurs during execution of a READ statement that contains neither an input/output status specifier nor an end-of-file specifier (13.7.2), execution of the executable program is terminated.

13.7 I/O STATUS, ERROR, AND END-OF-FILE SPECIFIERS

The form of an input/output status specifier is:

IOSTAT = ios

where ios is an integer variable or integer array element.

Execution of an input/output statement containing this specifier

causes ios to become defined:

(1) with a zero value if neither an error condition nor an end-of-file condition is encountered by the processor,

(2) with a processor-dependent positive integer value if an error condition is encountered, or

(3) with a processor-dependent negative integer value if an end-of-file condition is encountered and no error condition is encountered.

13.7.1 ERROR SPECIFIER.

The form of an error specifier is:

ERR = s

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

If an input/output statement contains an error specifier and the processor encounters an error condition during execution of the statement:

(1) execution of the input/output statement terminates,

(2) the position of the file specified in the input/output statement becomes indeterminate,

(3) if the input/output statement contains an input/output status specifier (13.7), the variable or array element ios becomes defined with a processor-dependent positive integer value, and

(4) execution continues with the statement labeled s.

13.7.2 END-OF-FILE SPECIFIER.

The form of an end-of-file specifier is:

END = s

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

If a READ statement contains an end-of-file specifier and the processor encounters an end-of-file condition and no error condition during execution of the statement:

(1) execution of the READ statement terminates,

(2) If the READ statement contains an input/output status specifier (13.7), the variable or array element *ios* becomes defined with a processor-dependent negative integer value, and

(3) execution continues with the statement labeled *s*.

13.8 READ, WRITE, PRINT, AND PUNCH STATEMENTS

The READ statement is the data transfer input statement. The WRITE, PRINT, and PUNCH statements are the data transfer output statements. The forms of the data transfer input/output statements are:

READ (*cilist*) [*iolist*]

READ *f* [,*iolist*]

WRITE (*cilist*) [*iolist*] [,]

PRINT *f* [,*iolist*] [,]

PUNCH *f* [,*iolist*] [,]

where: *cilist* is a control information list (13.8.1) that includes:

(1) A reference to the source or destination of the data to be transferred

(2) Optional specification of editing processes

(3) Optional specifiers that determine the execution sequence on the occurrence of certain events

(4) Optional specification to identify a record

(5) Optional specification to provide the return of the input/output status

f is a format identifier (13.4)

iolist is an input/output list (13.8.2) specifying the data to be transferred

If the format identifier *f* is an asterisk, the list *iolist* in a WRITE, PRINT or PUNCH statement may be followed by a comma. The comma has no effect. If the format identifier *f* is a NAMELIST group name (13.14), *iolist* must not be present.

13.8.1 CONTROL INFORMATION LIST.

A *control information list*, *cilist*, is a list (2.11) whose list items may be any of the following:

```
+----------------+
| [UNIT =] u     |
| [FMT = ] f     |
| REC = rn       |
| IOSTAT = ios   |
| ERR = s        |
| END = s        |
+----------------+
```

A control information list must contain exactly one unit specifier (13.3.3), at most one format specifier (13.4), at most one record specifier (13.5), at most one input/output status specifier (13.7), at most one error specifier (13.7.1), and at most one end-of-file specifier (13.7.2).

If the control information list contains a format specifier, the statement is a *formatted input/output statement*; otherwise, it is an *unformatted input/output statement*.

If the control information list contains a record specifier, the statement is a *direct access input/output statement*; otherwise, it is a *sequential access input/output statement*.

If the optional characters UNIT= are omitted from the unit specifier, the unit specifier must be the first item in the control information list.

If the optional characters FMT= are omitted from the format specifier, the format specifier must be the second item in the control information list and the first item must be the unit specifier without the optional characters UNIT=.

A control information list must not contain both a record specifier and an end-of-file specifier.

If the format identifier is an asterisk, the statement is a *list-directed input/output statement* and a record specifier must not be present.

If the format identifier is a NAMELIST group name, the statement is a *NAMELIST input/output statement* and a record specifier must not be present.

In a WRITE statement, the control information list must not contain an end-of-file specifier.

If the unit specifier specifies an internal file, the control information list must contain a format identifier other than an asterisk or a NAMELIST group name and must not contain a record specifier.

13.8.2 INPUT/OUTPUT LIST

An *input/output list*, *iolist*, specifies the entities whose values are transferred by a data transfer input/output statement.

An input/output list is a list (2.11) of input/output list items and implied-DO lists (13.8.2.3). An *input/output list item* is either an input list item or an output list item.

If an array name appears as an input/output list item, it is treated as if all of the elements of the array were specified in the order given by array element ordering (5.3.4). The name of an assumed-size dummy array must not appear as an input/output list item.

13.8.2.1 Input List Items.

An *input 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 section name

Only input list items may appear as input/output list items in an input statement.

13.8.2.2 Output List Items.

An *output 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 section name

(6) Any other expression except a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses unless the operand is the symbolic name of a constant

Note that a constant, an expression involving operators or function references, or an expression enclosed in parentheses may appear as an output list item but must not appear as an input list item.

13.8.2.3 Implied-DO List.

An implied-DO list is of the form:

(dlist, i = e1, e2 [,e3])

where: i, e1, e2, and e3 are as specified for the DO statement (11.10)

dlist is an input/output list

The range of an implied-DO list is the list dlist. Note that dlist may contain implied-DO lists. The iteration count and the values of the DO-variable i are established from e1, e2, and e3 exactly as for a DO-loop. In an input statement, the DO-variable i, or an associated entity, must not appear as an input list item in dlist. When an implied-DO list appears in an input/output list, The list items in dlist are specified once for each iteration of the implied-DO list with appropriate substitution of values for any occurrence of the DO-variable i.

13.9 EXECUTION OF A DATA TRANSFER INPUT/OUTPUT STATEMENT

The effect of executing a data transfer input/output statement must be as if the following operations were performed in the order specified:

(1) Determine the direction of data transfer

(2) Identify the unit

(3) Establish the format if any is specified

(4) Position the file prior to data transfer

(5) Transfer data between the file and the entities specified by the input/output list (if any) or identified by the NAMELIST group name

(6) Position the file after data transfer

(7) Cause the specified integer variable or array element in the

input/output status specifier (if any) to become defined

13.9.1 DIRECTION OF DATA TRANSFER

Execution of a READ statement causes values to be transferred from a file to the entities specified by the input list, if one is specified.

Execution of a WRITE, PRINT, or PUNCH statement causes values to be transferred to a file from the entities specified by the output list or NAMELIST group name and format specification (if any). Execution of a WRITE, PRINT, or PUNCH statement for a file that does not exist creates the file, unless an error condition occurs.

13.9.2 IDENTIFYING A UNIT

A data transfer input/output statement that contains a control information list (13.8.1) includes a unit specifier that identifies an external unit or an internal file. A READ statement that does not contain a control information list specifies a particular processor-determined unit, which is the same as the unit identified by an asterisk in a READ statement that contains a control information list. A PRINT statement specifies some other processor-determined unit, which is the same as the unit identified by an asterisk in a WRITE statement. A PUNCH statement specifies still another processor-determined unit. Thus each data transfer input/output statement identifies an external unit or an internal file.

Data transfer input/output statements that do not contain control information lists refer to units that are preconnected as follows:

Statement	Standard Unit	File Name
READ	L"INPUT"	'INPUT'
PRINT	L"OUTPUT"	'OUTPUT'
PUNCH	L"PUNCH"	'PUNCH'

The unit identified by a data transfer input/output statement must be connected to a file when execution of the statement begins.

Data transfer statements that do not contain a control information list refer to preconnected files that are processor defined.

13.9.3 ESTABLISHING A FORMAT

If the control information list contains a format identifier other than an asterisk or NAMELIST group name, the format specification identified by the format identifier is established. If the format

identifier is an asterisk, list-directed formatting is established. If the format identifier is a NAMELIST group name, NAMELIST formatting (14.7) is established.

On output, if an internal file has been specified, a format specification (14.1) that is in the file or is associated (18.1) with the file must not be specified.

13.9.4 FILE POSITION PRIOR TO DATA TRANSFER

The positioning of the file prior to data transfer depends on the method of access: sequential or direct.

If the file contains an endfile record, the file must not be positioned after the endfile record prior to data transfer.

13.9.4.1 Sequential Access

On input, the file is positioned at the beginning of the next record. This record becomes the current record. On output, a new record is created and becomes the last record of the file.

An internal file is always positioned at the beginning of the first record of the file. This record becomes the current record.

13.9.4.2 Direct Access

For direct access, the file is positioned at the beginning of the record specified by the record specifier (13.5). This record becomes the current record.

13.9.5 DATA TRANSFER

Data are transferred between records and entities specified by the input/output list. The list items are processed in the order of the input/output list.

All values needed to determine which entities are specified by an input/output list item are determined at the beginning of the processing of that item.

All values are transmitted to or from the entities specified by a list item prior to the processing of any succeeding list item. In the example,

```
READ (3) N, A(N)
```

two values are read; one is assigned to N, and the second is assigned to A(N) for the new value of N.

An input list item, or an entity associated with it (18.1.3), must

not contain any portion of the established format specification.

If an internal file has been specified, an input/output list item must not be in the file or associated with the file.

A DO-variable becomes defined at the beginning of processing of the items that constitute the range of an implied-DO list.

On output, every entity whose value is to be transferred must be defined.

On input, an attempt to read a record of a file connected for direct access that has not previously been written causes all entities specified by the input list to become undefined.

13.9.5.1 Unformatted Data Transfer

During unformatted data transfer, data are transferred without editing between the current record and the entities specified by the input/output list. Exactly one record is read or written.

On input, the file must be positioned so that the record read is an unformatted record or an endfile record.

On input, the number of values required by the input list must be less than or equal to the number of values in the record.

On input, the type of each value in the record must agree with the type of the corresponding entity in the input list, except that one complex value may correspond to two real list entities or two real values may correspond to one complex list entity. If an entity in the input list is of type character, the length of the character entity must agree with the length of the character value.

Also, a Boolean value may correspond to an integer or real list entity, or to either half of a double precision or complex list entity; and an integer or real value, or either half of a double precision or complex value, may correspond to a Boolean list entity.

On output to a file connected for direct access, the output list must not specify more values than can fit into a record.

On output, if the file is connected for direct access and the values specified by the output list do not fill the record, the remainder of the record is undefined.

If the file is connected for formatted input/output, unformatted data transfer is prohibited.

The unit specified must be an external unit.

13.9.5.2 Formatted Data Transfer

During formatted data transfer, data are transferred with editing between the entities specified by the input/output list and the file. The current record and possibly additional records are read or written.

On input, the file must be positioned so that the record read is a formatted record or an endfile record.

If the file is connected for unformatted input/output, formatted data transfer is prohibited.

13.9.5.2.1 USING A FORMAT SPECIFICATION

If a format specification has been established, format control (14.3) is initiated and editing is performed as described in 14.3 through 14.5.

On input, the input list and format specification must not require more characters from a record than the record contains.

If the file is connected for direct access, the record number is increased by one as each succeeding record is read or written.

On output, if the file is connected for direct access or is an internal file and the characters specified by the output list and format do not fill a record, blank characters are added to fill the record.

On output, if the file is connected for direct access or is an internal file, the output list and format specification must not specify more characters for a record than can fit into the record.

13.9.5.2.2 LIST-DIRECTED FORMATTING

If list-directed formatting has been established, editing is performed as described in 14.6.

13.9.5.2.3 PRINTING OF FORMATTED RECORDS

The transfer of information in a formatted record to certain devices determined by the processor is called _printing_. 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.

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

Character	Vertical Spacing Before Printing
Blank	One Line
0	Two Lines
1	To First Line of Next Page
+	No Advance

If there are no characters in the record (14.5.4), the vertical spacing is one line and no characters other than blank are printed in that line.

A PRINT statement does not imply that printing will occur, and a WRITE statement does not imply that printing will not occur.

13.9.6 FILE POSITION AFTER DATA TRANSFER

If an end-of-file condition exists as a result of reading an endfile record, the file is positioned after the endfile record.

If no error condition or end-of-file condition exists, the file is positioned after the last record read or written and that record becomes the preceding record. A record written on a file connected for sequential access becomes the last record of the file.

If the file is positioned after the endfile record, execution of a data transfer input/output statement is prohibited. However, a BACKSPACE or REWIND statement may be used to reposition the file.

If an error condition exists, the position of the file is indeterminate.

13.9.7 INPUT/OUTPUT STATUS SPECIFIER DEFINITION

If the data transfer input/output statement contains an input/output status specifier, the integer variable or array element ios becomes defined. If no error condition or end-of-file condition exists, the value of ios is zero. If an error condition exists, the value of ios is positive. If an end-of-file condition exists and no error condition exists, the value of ios is negative.

13.10 AUXILIARY INPUT/OUTPUT STATEMENTS

13.10.1 OPEN STATEMENT

An OPEN statement may be used to connect (13.3.2) an existing file to a unit, create a file (13.2.1) that is preconnected, create a file and connect it to a unit, or change certain specifiers of a connection between a file and a unit.

The form of an OPEN statement is:

OPEN (olist)

where olist is a list (2.11) of specifiers:

```
+---------------+
| [UNIT =] u    |
| IOSTAT = ios  |
| ERR = s       |
| FILE = fin    |
| STATUS = sta  |
| ACCESS = acc  |
| FORM = fm     |
| RECL = rl     |
| BLANK = blnk  |
| BUFL = bl     |
+---------------+
```

olist must contain exactly one external unit specifier (13.3.3) and may contain at most one of each of the other specifiers.

The other specifiers are described as follows:

IOSTAT = ios

is an input/output status specifier (13.7). Execution of an OPEN statement containing this specifier causes ios to become defined with a zero value if no error condition exists or with a processor-dependent positive integer value if an error condition exists.

ERR = s

is an error specifier (13.7.1).

FILE = fin

fin is a character scalar expression whose value when any trailing blanks are removed is the name of the file to be connected to the specified unit. The alphabetic case of letters in the value of the character expression is not significant. The file name must be a name that is allowed by the processor. If this specifier is omitted and the unit is not connected to a file, it becomes connected to a processor-determined file. (See also 13.10.1.1.)

The processor determines a file name from the unit specifier u as follows

If INT(u) has a value representable by the digit string n in the

range 0-999, the file name is TAPEn.

If BOOL(u) has a value of the form L"f", where f is a valid system file name starting with a letter, the file name is f except in two cases: if BOOL(u) has the value L"INPUT" the file name is INPUT, and if BOOL(u) has the value L"OUTPUT" the file name is OUTPUT. Upper and lower case letters in the file name are interpreted as variants of the same letter.

Otherwise, the unit specified does not exist.

STATUS = sta

sta is a scalar character expression whose value when any trailing blanks are removed is OLD, NEW, SCRATCH, or UNKNOWN. The alphabetic case of letters in the value of the character expression is not significant. If OLD or NEW is specified, a FILE= specifier must be given. If OLD is specified, the file must exist. If NEW is specified, the file must not exist. Successful execution of an OPEN statement with NEW specified creates the file and changes the status to OLD (13.10.1.1). 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 (13.2.1) at the execution of a CLOSE statement referring to the same unit or at the termination of the executable program. SCRATCH must not be specified with a named file. If UNKNOWN is specified, the status is processor dependent. If this specifier is omitted, a value of UNKNOWN is assumed.

ACCESS = acc

acc is a scalar character expression whose value when any trailing blanks are removed is SEQUENTIAL or DIRECT. The alphabetic case of letters in the value of the character expression is not significant. It specifies the access method for the connection of the file as being sequential or direct (13.2.4). If this specifier is omitted, the assumed value is SEQUENTIAL. For an existing file, the specified access method must be included in the set of allowed access methods for the file (13.2.4). For a new file, the processor creates the file with a set of allowed access methods that includes the specified method.

FORM = fm

fm is a scalar character expression whose value when any trailing blanks are removed is FORMATTED, UNFORMATTED, or BUFFERED. The alphabetic case of letters in the value of the character expression is not significant. It specifies that the file is being connected for formatted, unformatted, or

buffered input/output, respectively. If this specifier is omitted, a value of UNFORMATTED is assumed if the file is being connected for direct access, and a value of FORMATTED is assumed if the file is being connected for sequential access. For an existing file, the specified form must be included in the set of allowed forms for the file (13.2.2). For a new file, the processor creates the file with a set of allowed forms that includes the specified form.

RECL = rl

rl is a scalar integer expression whose value must be positive. It specifies the length of each record in a file being connected for direct access or for formatted sequential access. If the file is being connected for formatted input/output, the length is the number of characters. If the file is being connected for unformatted direct access input/output, the length is measured in processor-dependent units. For an existing file, the value of rl must be included in the set of allowed record lengths for the file (13.2.2). For a new file, the processor creates the file with a set of allowed record lengths that includes the specified value. This specifier must be given when a file is being connected for direct access.

BLANK = blnk

blnk is a scalar character expression whose value when any trailing blanks are removed is NULL or ZERO. The alphabetic case of letters in the value of the character expression is not significant. If NULL 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, a value of NULL is assumed. This specifier is permitted only for a file being connected for formatted input/output.

The unit specifier is required to appear; all other specifiers are optional, except that the record length rl must be specified if a file is being connected for direct access. Note that some of the specifications have an assumed value if they are omitted.

The unit specified must exist.

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 the executable program.

BUFL=bl

bl is an integer or Boolean expression. The value of INT(*bl*) must be nonnegative and specifies the buffer length (15.1.1).

13.10.1.1 OPEN of a Connected Unit.

If a unit is connected to a file that exists, execution of an OPEN statement for that unit is permitted. If the FILE= specifier is not included in the OPEN statement, the file to be connected to the unit is the same as the file to which the unit is connected.

If the file to be connected to the unit does not exist, but is the same as the file to which the unit is preconnected, the properties specified by the OPEN statement become a part of the connection.

If the file to be connected to the unit is not the same as the file to which the unit is connected, the effect is as if a CLOSE statement (13.10.2) without a STATUS= specifier had been executed for the unit immediately prior to the execution of the OPEN statement.

If the file to be connected to the unit is the same as the file to which the unit is connected, only the BLANK= specifier may have a value different from the one currently in effect. Execution of the OPEN statement causes the new value of the BLANK= specifier to be in effect. The position of the file is unaffected.

If a file is connected to a unit, execution of an OPEN statement on that file and a different unit is permitted. The effect is that the file becomes connected to more than one unit.

13.10.2 CLOSE STATEMENT.

A CLOSE statement is used to terminate the connections of a particular file to the unit or units to which it is connected.

The form of a CLOSE statement is:

CLOSE (*cllist*)

where *cllist* is a list (2.11) of specifiers:

```
+----------------+
! [UNIT =] u     !
! IOSTAT = ios   !
! ERR = s        !
! STATUS = sta   !
+----------------+
```

cllist must contain exactly one external unit specifier (13.3.3) and may contain at most one of each of the other specifiers.

The other specifiers are described as follows:

IOSTAT = ios

> is an input/output status specifier (13.7). Execution of a CLOSE statement containing this specifier causes ios to become defined with a zero value if no error condition exists or with a processor-dependent positive integer value if an error condition exists.

ERR = s

> is an error specifier (13.7.1)

STATUS = sta

> sta is a scalar character expression whose value when any trailing blanks are removed is KEEP or DELETE. The alphabetic case of letters in the value of the character expression is not significant. sta determines the disposition of the file that is connected to the specified unit. KEEP must not be specified for a file whose status prior to execution of the CLOSE statement is SCRATCH. If KEEP is specified for a file that exists, the file continues to exist after the execution of the CLOSE statement. If KEEP is specified for a file that does not exist, the file will not exist after the execution of the CLOSE statement. If DELETE is specified, the file will not exist after execution of the CLOSE statement. If this specifier is omitted, the assumed value is KEEP, unless the file status prior to execution of the CLOSE statement is SCRATCH, in which case the assumed value is DELETE.

Execution of a CLOSE statement that refers to a unit may occur in any program unit of an executable program and need not occur in the same program unit as the execution of an OPEN statement referring to that unit.

Execution of a CLOSE statement specifying a unit that does not exist or has no file connected to it is permitted and affects no file.

After a unit has been disconnected by execution of a CLOSE statement, it may be connected again within the same executable program, either to the same file or to 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, either to the same unit or to a different unit, provided that the file still exists.

If a file is connected to more than one unit, the CLOSE statement may refer to any one of these units with the same effect: each unit

is disconnected.

13.10.2.1 Implicit Close at Termination of Execution.

At termination of execution of an executable program for reasons other than an error condition, all units that are connected are closed. Each unit is closed with status KEEP unless the file status prior to termination of execution was SCRATCH, in which case the unit is closed with status DELETE. Note that the effect is as though a CLOSE statement without a STATUS= specifier were executed on each connected unit.

13.10.3 INQUIRE STATEMENT.

An INQUIRE statement may be used to inquire about properties of a particular named file or of the connection to a particular unit. There are two forms of the INQUIRE statement: inquire by file and inquire by unit. All value assignments are done according to the rules for assignment statements.

The INQUIRE statement may be executed before, while, or after a file is connected to a unit. All values assigned by the INQUIRE statement are those that are current at the time the statement is executed.

13.10.3.1 INQUIRE by File.

The form of an INQUIRE by file statement is:

```
INQUIRE (iflist)
```

where *iflist* is a list (2.11) of specifiers that must contain exactly one file specifier and may contain other inquiry specifiers. The *iflist* may contain at most one of each of the inquiry specifiers described in 13.10.3.3.

The form of a file specifier is:

```
FILE = fin
```

where *fin* is a character expression whose value when any trailing blanks are removed specifies the name of the file being inquired about. The named file need not exist or be connected to a unit. The value of *fin* must be of a form acceptable to the processor as a file name.

13.10.3.2 INQUIRE by Unit.

The form of an INQUIRE by unit statement is:

```
INQUIRE (iulist)
```

where iulist is a list (2.11) of specifiers that must contain exactly one external unit specifier (13.3.3) and may contain other inquiry specifiers. The iulist may contain at most one of each of the inquiry specifiers described in 13.10.3.3. The unit specified need not exist or be connected to a file. If it is connected to a file, the inquiry is being made about the connection and about the file connected.

13.10.3.3 Inquiry Specifiers.

The following inquiry specifiers may be used in either form of the INQUIRE statement:

```
+---------------------+
| IOSTAT = ios        |
| ERR = s             |
| EXIST = ex          |
| OPENED = od         |
| NUMBER = num        |
| NAMED = nmd         |
| NAME = fn           |
| ACCESS = acc        |
| SEQUENTIAL = seq    |
| DIRECT = dir        |
| FORM = fm           |
| FORMATTED = fmt     |
| UNFORMATTED = unf   |
| RECL = rcl          |
| NEXTREC = nr        |
| BLANK = blnk        |
+---------------------+
```

The specifiers are described as follows:

IOSTAT = ios

is an input/output status specifier (13.7). Execution of an INQUIRE statement containing this specifier causes ios to become defined with a zero value if no error condition exists or with a processor-dependent positive integer value if an error condition exists.

ERR = s

is an error specifier (13.7.1)

EXIST = ex

ex is a logical variable or logical array element. Execution of an INQUIRE by file statement causes ex to be assigned the value true if there exists a file with the specified name;

otherwise, ex is assigned the value false. Execution of an INQUIRE by unit statement causes ex to be assigned the value true if the specified unit exists; otherwise, ex is assigned the value false.

OPENED = od

od is a logical variable or logical array element. Execution of an INQUIRE by file statement causes od to be assigned the value true if the file specified is connected to a unit; otherwise, od is assigned the value false. Execution of an INQUIRE by unit statement causes od to be assigned the value true if the specified unit is connected to a file; otherwise, od is assigned the value false.

NUMBER = num

num is an integer variable or integer array element that is assigned the value of the external unit identifier of a unit that is currently connected to the file. If more than one unit is currently connected to the file, the choice of the unit used to assign a value to num is as described below. If there is an external unit identifier u currently connected to the file such that either

(1) INT(u) has a value in the range 0-999, or

(2) BOOL(u) has a value of the form L"TAPEk", where k is an integer in the range 0-999 with no leading zero,

then the value assigned to num will be in the range 0-999. (In case (1) the value assigned to num is the value of u, and in case (2) it is the value of k. If external unit identifiers of both types (1) and (2) are currently connected to the file, the processor may assign either a type (1) or a type (2) value.) Otherwise, the value assigned to num is of the form INT(L"f"), where f is a valid system file name.

NAMED = nmd

nmd is a logical variable or logical array element that is assigned the value true if the file has a name; otherwise, it is assigned the value false.

NAME = fn

fn is a character variable or character array element that is assigned the value of the name of the file, if the file has a name; otherwise, it becomes undefined. If the name of the file contains alphabetic characters they will be returned in upper case. Note that if this specifier appears in an

INQUIRE by file statement, its value is not necessarily the same as the name given in the FILE= specifier. For example, the processor may return a file name qualified by a user identification. However, the value returned must be suitable for use as the value of a FILE= specifier in an OPEN statement.

ACCESS = acc

acc is a character variable or character array element that is assigned the value SEQUENTIAL if the file is connected for sequential access, and DIRECT if the file is connected for direct access. If there is no connection, acc becomes undefined.

SEQUENTIAL = seq

seq is a character variable or character array element that is assigned the value YES if SEQUENTIAL is included in the set of allowed access methods for the file, NO if SEQUENTIAL is not included in the set of allowed access methods for the file, and UNKNOWN if the processor is unable to determine whether or not SEQUENTIAL is included in the set of allowed access methods for the file.

DIRECT = dir

dir is a character variable or character array element that is assigned the value YES if DIRECT is included in the set of allowed access methods for the file, NO if DIRECT is not included in the set of allowed access methods for the file, and UNKNOWN if the processor is unable to determine whether or not DIRECT is included in the set of allowed access methods for the file.

FORM = fm

fm is a character variable or character array element that is assigned the value FORMATTED if the file is connected for formatted input/output, UNFORMATTED if the file is connected for unformatted input/output, and BUFFERED if the file is connected for buffered input/output. If there is no connection, fm becomes undefined.

FORMATTED = fmt

fmt is a character variable or character array element that is assigned the value YES if FORMATTED is included in the set of allowed forms for the file, NO if FORMATTED is not included in the set of allowed forms for the file, and UNKNOWN if the processor is unable to determine whether or

not FORMATTED is included in the set of allowed forms for the file.

UNFORMATTED = unf

unf is a character variable or character array element that is assigned the value YES if UNFORMATTED is included in the set of allowed forms for the file, NO if UNFORMATTED is not included in the set of allowed forms for the file, and UNKNOWN if the processor is unable to determine whether or not UNFORMATTED is included in the set of allowed forms for the file.

RECL = rcl

rcl is an integer variable or integer array element that is assigned the value of the record length of the file connected for direct access. If the file is connected for formatted input/output, the length is the number of characters. If the file is connected for unformatted input/output, the length is measured in processor-dependent units. If there is no connection, rcl becomes undefined.

NEXTREC = nr

nr is an integer variable or integer array element that is assigned the value n+1, where n is the record number of the last record read or written on the file connected for direct access. If the file is connected but no records have been read or written since the connection, nr is assigned the value 1. If the file is not connected for direct access or if the position of the file is indeterminate because of a previous error condition, nr becomes undefined.

BLANK = blnk

blnk is a character variable or character array element that is assigned the value NULL if null blank control is in effect for the file connected for formatted input/output, and is assigned the value ZERO if zero blank control is in effect for the file connected for formatted input/output. If there is no connection, or if the connection is not for formatted input/output, blnk becomes undefined.

A variable or array element that is used as a specifier in an INQUIRE statement, or any associated entity, must not be referenced by any other specifier in the same INQUIRE statement.

Execution of an INQUIRE by file statement causes the specifier variables or array elements nmd, fn, seq, dir, fmt, and unf to be

assigned values 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. Note that num becomes defined if and only if od becomes defined with the value true. Note also that the specifier variables or array elements acc, fm, rcl, nr, and blnk may become defined only if od becomes defined with the value true.

Execution of an INQUIRE by unit statement causes the specifier variables or array elements num, nmd, fn, acc, seq, dir, fm, fmt, unf, rcl, nr, and blnk to be assigned values only if the specified unit exists and if a file is connected to the unit; otherwise, they become undefined.

If an error condition occurs during execution of an INQUIRE statement, all of the inquiry specifier variables and array elements except ios become undefined.

Note that the specifier variables or array elements ex and od always become defined unless an error condition occurs.

13.10.4 FILE_POSITIONING_STATEMENTS.

The forms of the file positioning statements are:

```
BACKSPACE u

BACKSPACE (alist)

ENDFILE u

ENDFILE (alist)

REWIND u

REWIND (alist)
```

where: u is an external unit identifier (13.3.3)

alist is a list (2.11) of specifiers:

```
+-------------+
! [UNIT =] u  !
! IOSTAT = ios !
! ERR = s     !
+-------------+
```

alist must contain exactly one external unit specifier (13.3.3) and may contain at most one of each of the other specifiers.

The external unit specified by a BACKSPACE, ENDFILE, or REWIND statement must be connected for sequential access.

Execution of a file positioning statement containing an input/output status specifier causes _ios_ to become defined with a zero value if no error condition exists or with a processor-dependent positive integer value if an error condition exists.

13.10.4.1 BACKSPACE Statement.

Execution of a BACKSPACE statement causes the file connected to the specified unit to be positioned before the preceding record. If there is no preceding record, the position of the file is not changed. Note that if the preceding record is an endfile record, the file becomes positioned before the endfile record.

Backspacing a file that is connected but does not exist is prohibited.

Backspacing over records written using list-directed formatting is prohibited.

13.10.4.2 ENDFILE Statement.

Execution of an ENDFILE statement writes an endfile record as the next record of the file. The file is then positioned after the endfile record. If the file may also be connected for direct access, only those records before the endfile record are considered to have been written. Thus, only those records may be read during subsequent direct access connections to the file.

After execution of an ENDFILE statement, a BACKSPACE or REWIND statement must be used to reposition the file prior to execution of any data transfer input/output statement.

Execution of an ENDFILE statement for a file that is connected but does not exist creates the file.

13.10.4.3 REWIND Statement.

Execution of a REWIND statement causes the specified file to be positioned at its initial point. Note that if the file is already positioned at its initial point, execution of this statement has no effect on the position of the file.

Execution of a REWIND statement for a file that is connected but does not exist is permitted but has no effect.

13.11 RESTRICTIONS ON FUNCTION REFERENCES AND LIST ITEMS

A function must not be referenced within an expression appearing anywhere in an input/output statement if such a reference causes an input/output statement to be executed. Note that a restriction in the evaluation of expressions (6.6) prohibits certain side effects.

13.12 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 must not refer to the unit.

13.13 NAMELIST INPUT/OUTPUT

NAMELIST provides formatted input/output with processor-determined editing (14.7).

A _NAMELIST group name_ provides the means of referring to a NAMELIST input/output list. Usage of a group name is the means of specifying NAMELIST formatting. A NAMELIST statement is used to specify a NAMELIST group name and the input/output list to be subsequently associated with that group name.

NAMELIST formatting is established for an input/output data transfer by using a NAMELIST group name as the format identifier _f_ in a READ, WRITE, PRINT, or PUNCH statement (13.4 (6)). The statement must not include an input/output list.

13.13.1 NAMELIST STATEMENT

The form of a NAMELIST statement is:

NAMELIST /_grpname_/ _niolist_ [/_grpname_/ _niolist_]...

where: _grpname_ is a NAMELIST group name. Only one appearance of a group name in all of the NAMELIST statements of a program unit is permitted. A group name of END is prohibited.

niolist is a NAMELIST input/output list of one or more items, each of which must be one of the following:

(1) A variable name

(2) An array name

(3) A virtual (IDENTIFY) array name

Each name in the list _niolist_ may be of any data type. _niolist_ may not contain the name of an assumed size array.

13.13.2 NAMELIST_DATA_TRANSFER

A NAMELIST_block is one or more formatted records that consist of a sequence of characters in NAMELIST format (14.7). Execution of an input/output data transfer statement with NAMELIST formatting causes one NAMELIST block to be transferred.

Execution of a WRITE, PRINT, or PUNCH statement with NAMELIST formatting causes one NAMELIST block to be written to a file. Data are transferred from internal storage in the order specified by the input/output list associated with the NAMELIST group name that appears in the output data transfer statement.

Execution of a READ statement with NAMELIST formatting causes one NAMELIST block to be read from a file. The NAMELIST group name in the block read must be the same as the group name in the READ statement being executed. Each variable or array name in the block must appear in the input/output list associated with the group name. Item names in the block may occur in any order and number. Note that an item name may appear more than once in a block, possibly resulting in more than one definition of an entity. The block is transferred with NAMELIST editing (14.7) to internal storage in the order of item name appearance. Values are transmitted to the entities specified by the item names. The definition status and value of each entity whose name does not appear in the NAMELIST block are unchanged upon completion of the transfer. Note that an entity named in the associated input/output list but not named in the NAMELIST block retains its prior definition status and value.

On input, an error condition exists if the file is not positioned at the beginning of a NAMELIST block. If the current NAMELIST block is not the one specified in the READ statement, the input file will be positioned forward to the NAMELIST block corresponding to the NAMELIST group name in the READ statement. If the file is initially positioned preceding an endfile record, an end-of-file condition occurs and the actions specified in section 13.6 take place. If an endfile record is encountered while reading or while positioning forward searching for a NAMELIST block, an error condition exists and the actions specified in section 13.6 take place.

The effect of executing a data transfer input/output statement with NAMELIST formatting is otherwise as described in Section 13.9.

13.14 ENCODE_AND_DECODE_STATEMENTS

The ENCODE statement is the extended internal file (13.2.5) data transfer output statement. The DECODE statement is the extended internal file data transfer input statement. The forms of the statements are:

ENCODE (k, f, u) [iolist]

DECODE (k, f, u) [iolist]

where:

k — is an unsigned integer constant or integer variable having a positive, nonzero value. The value specifies the number of characters to be transferred to or from each record of the file identified by u.

f — is a format specifier (13.14) which is specified as one of the following:

(1) The statement label of a FORMAT statement that appears in the same program unit as the format identifier

(2) A character array name or character array section (14.1.2)

u — is an internal file identifier (13.3.3) for an extended internal file

iolist — is an input/output list (13.8.2) specifying the data to be transferred

Execution of an ENCODE statement causes values to be transferred to an extended internal file from the entities specified by the output list iolist (if any) and the format identifier f. The execution sequence, restrictions and error conditions are as described in Section 13.9 for a formatted WRITE statement that transfers data to an internal file.

Execution of a DECODE statement causes values to be transferred from an extended internal file to the entities specified by the input list iolist (if any). Execution proceeds as described in Section 13.9 for a formatted READ statement that transfers data from an internal file.

The length of each record of an extended internal file is established by the ENCODE statement that causes the file to exist by defining or redefining it. Note that the record length may be changed by subsequent redefinition. The record length is measured in characters and is the number of characters transferred. It is g*((k+7)/g) characters, where g is the maximum number of characters that can be represented by the processor in one numeric storage unit.

Action is unspecified if the total length of all records read or written exceeds the number of numeric storage units of the file (13.2.5.3).

Action is unspecified if any element of iolist is in the file or is associated (18.1) with the file.

On output, a format specification (14.1) that is in the file or is associated (18.1) with the file must not be specified.

Note that an extended internal file may be defined or redefined by means other than an ENCODE statement, such as a Boolean assignment statement. Such means must ensure that the record length is established as provided above and in Section 13.2.5.3.

13.15 BUFFER IN AND BUFFER OUT STATEMENTS

The BUFFER IN and BUFFER OUT statements initiate unformatted data transfer. Program execution may continue immediately while the transfer proceeds.

The forms of the statements are:

```
BUFFER IN  (u,p) (s1,sn)

BUFFER OUT (u,p) (s1,sn)
```

where:

- u is an external unit identifier (13.3.3) which is not an asterisk
- p is an integer constant or an integer variable name. The value of p is not significant
- s1 is a variable name or an array element name of type other than character, bit or half precision. s1 specifies the first entity of a storage sequence (18.1.1) to be transferred.
- sn is a variable name or an array element name of type other than character, bit or half precision. sn specifies the last, or possibly only, entity of a storage sequence to be transferred.

Execution of a BUFFER IN statement initiates the transfer of values from one record of an unformatted external file to the storage sequence s1...sn. Execution of a BUFFER OUT statement initiates the transfer of values from the storage sequence s1...sn to one record of an unformatted external file. The entities s1 and sn must be the same variable or must be in the same array, common or equivalence class. Action is undefined if one of these relationships does not hold.

The execution sequence, restrictions and error conditions are as

described in Section 13.9 for an input/output data transfer with an unformatted external file, except

(1) Immediately after the data transfer has been successfully initiated, the processor resumes normal program execution.

(2) On input, the storage sequence s1...sn can be longer than the number of values in the record. The definition status of the remaining part of the sequence is not changed.

A BUFFER IN or BUFFER OUT data transfer is completed by referencing the processor-supplied function UNIT (16.11.6). Action is unspecified if the program references or defines the storage sequence s1...sn during an input data transfer, or if the program redefines the sequence or causes it to become undefined during an output data transfer.

14.0 FORMAT SPECIFICATION

A format used in conjunction with formatted input/output statements provides information that directs the editing between the internal representation and the character strings of a record or a sequence of records in the file.

A format specification provides explicit editing information. An asterisk (*) as a format identifier in an input/output statement indicates list-directed formatting (14.6).

14.1 FORMAT SPECIFICATION METHODS

Format specifications may be given:

(1) In FORMAT statements

(2) As values of character arrays, character variables, or other character expressions

14.1.1 FORMAT STATEMENT.

The form of a FORMAT statement is:

FORMAT *fs*

where *fs* is a format specification, as described in 14.2. The statement must be labeled.

14.1.2 CHARACTER FORMAT SPECIFICATION.

If the format identifier (13.4) in a formatted input/output statement is a character array name, character variable name, or other character expression, the leftmost character positions of the specified entity must be in a defined state with character data that constitute a format specification when the statement is executed.

A character format specification must be of the form described in 14.2. Note that the form begins with a left parenthesis and ends with a right parenthesis. Character data may follow the right parenthesis that ends the format specification, with no effect on the format specification. Blank characters may precede the format specification.

If the format identifier is a character array name, the length of the format specification may exceed the length of the first element of the array; a character array format specification is considered to be a concatenation of all the array elements of the array in the order given by array element ordering (5.3.4). However, if a character array element name is specified as a format identifier,

CONTROL DATA PRIVATE

the length of the format specification must not exceed the length of the array element.

If the format identifier (13.4) in a formatted input-output statement is a noncharacter array name, the first m elements of the array must be in a defined state such that the juxtaposition of the values contained in the storage sequence of the first m elements of the array (for some positive integer m), with each numeric storage unit in the storage sequence interpreted as a Boolean value, constitutes a valid format specification when the statement is executed.

A noncharacter array format specification must be of the form described in 14.2. Note that the form begins with a left parenthesis and ends with a right parenthesis. There is no requirement on the information contained in the array following the right parenthesis that ends the format specification. Blank characters may precede the format specification.

14.2 FORM OF A FORMAT SPECIFICATION

The form of a _format specification_ is:

([_flist_])

where _flist_ is a list (2.11). The forms of the _flist_ items are:

[_r_] _ed_

ned

[_r_] _fs_

where: _ed_ is a repeatable edit descriptor (14.2.1)

ned is a nonrepeatable edit descriptor (14.2.1)

_fs_ is a format specification with a non-empty list _flist_

_r_ is a nonzero, unsigned, integer constant called a _repeat specification_

The comma used to separate list items in the list _flist_ may be omitted as follows:

(1) Between a P edit descriptor and an immediately following F, E, D, or G edit descriptor (14.5.9)

(2) Before or after a slash edit descriptor (14.5.4)

(3) Before or after a colon edit descriptor (14.5.5)

14.2.1 EDIT DESCRIPTORS

An *edit descriptor* is either a repeatable edit descriptor or a nonrepeatable edit descriptor.

The forms of a *repeatable edit descriptor* are:

I*w*
I*w*.*m*
F*w*.*d*
E*w*.*d*
E*w*.*d*E*e*
D*w*.*d*
G*w*.*d*
G*w*.*d*E*e*
L*w*
A
A*w*
O*w*
O*w*.*m*
Z*w*
Z*w*.*m*
B*w*
R*w*

where: I, F, E, D, G, L, A, O, Z, B, and R indicate the manner of editing

w and *e* are nonzero, unsigned, integer constants where *e* cannot exceed the value 6.

d and *m* are unsigned integer constants

The forms of a *nonrepeatable edit descriptor* are:

'*h*1 *h*2 ... *h*n'
*n*H*h*1*h*2 ... *h*n
"*h*1*h*2 ... *h*n"
T*c*
TL*c*
TR*c*
*n*X
/
:
S
SP
SS
*k*P
BN
BZ

where: apostrophe, H, quote, T, TL, TR, X, slash, colon, S, SP, SS, P, BN, and BZ indicate the manner of editing

h is one of the characters capable of representation by the processor

n and *c* are nonzero, unsigned, integer constants

k is an optionally signed integer constant

14.3 INTERACTION BETWEEN INPUT/OUTPUT LIST AND FORMAT

The beginning of formatted data transfer using a format specification (13.9.5.2.1) initiates *format control*. Each action of format control depends on information jointly provided by:

(1) the next edit descriptor contained in the format specification, and

(2) the next item in the input/output list, if one exists.

If an input/output list specifies at least one list item, at least one repeatable edit descriptor must exist in the format specification. Note that an empty format specification of the form ( ) may be used only if no list items are specified; in this case, one input record is skipped or one output record containing no characters is written. Except for an edit descriptor preceded by a repeat specification, *r ed*, and a format specification preceded by a repeat specification, *r(flist)*, a format specification is interpreted from left to right. A format specification or edit descriptor preceded by a repeat specification *r* is processed as a list of *r* format specifications or edit descriptors identical to the format specification or edit descriptor without the repeat specification. Note that an omitted repeat specification is treated the same as a repeat specification whose value is one.

To each repeatable edit descriptor interpreted in a format specification, there corresponds one item specified by the input/output list (13.8.2), except that a list item of type complex requires the interpretation of two F, E, D, or G edit descriptors. To each P, X, T, TL, TR, S, SP, SS, H, BN, BZ, slash, colon, quote, or apostrophe edit descriptor, there is no corresponding item specified by the input/output list, and format control communicates information directly with the record.

Whenever format control encounters a repeatable edit descriptor in a format specification, it determines whether there is a corresponding item specified by the input/output list. If there is such an item, it transmits appropriately edited information between the item and the record, and then format control proceeds. If there is no corresponding item, format control terminates.

If format control encounters a colon edit descriptor in a format specification and another list item is not specified, format control terminates.

If format control encounters the rightmost parenthesis of a complete format specification and another list item is not specified, format control terminates. However, if another list item is specified, the file is positioned at the beginning of the next record and format control then reverts to the beginning of the format specification terminated by the last preceding right parenthesis. If there is no such preceding right parenthesis, format control reverts to the first left parenthesis of the format specification. If such reversion occurs, the reused portion of the format specification must contain at least one repeatable edit descriptor. If format control reverts to a parenthesis that is preceded by a repeat specification, the repeat specification is reused. Reversion of format control, of itself, has no effect on the scale factor (14.5.7), the S, SP, or SS edit descriptor sign control (14.5.6), or the BN or BZ edit descriptor blank control (14.5.8).

14.4 POSITIONING BY FORMAT CONTROL

After each I, F, E, D, G, L, A, H, O, B, Z, R, quote, or apostrophe edit descriptor is processed, the file is positioned after the last character read or written in the current record.

After each T, TL, TR, X, or slash edit descriptor is processed, the file is positioned as described in 14.5.3 and 14.5.4.

If format control reverts as described in 14.3, the file is positioned in a manner identical to the way it is positioned when a slash edit descriptor is processed (14.5.4).

During a read operation, any unprocessed characters of the record are skipped whenever the next record is read.

14.5 EDITING

Edit descriptors are used to specify the form of a record and to direct the editing between the characters in a record and internal representations of data.

A *field* is a part of a record that is read on input or written on output when format control processes one I, F, E, D, G, L, A, H, O, B, Z, R, quote, or apostrophe edit descriptor. The *field width* is the size in characters of the field.

The internal representation of a datum corresponds to the internal representation of a constant of the corresponding type (Section 4).

14.5.1 APOSTROPHE AND QUOTE EDITING

The apostrophe edit descriptor has the form of a character constant. It causes characters to be written from the enclosed characters (including blanks) of the edit descriptor itself. An apostrophe edit descriptor must not be used on input.

The width of the field is the number of characters contained in, but not including, the delimiting apostrophes. Within the field, two consecutive apostrophes with no intervening blanks are counted as a single apostrophe.

The quote edit descriptor causes characters to be written from the enclosed characters (including blanks) of the edit descriptor itself. A quote edit descriptor must not be used on input.

The width of the field is the number of characters contained in, but not including, the delimiting quotes. Within the field, two consecutive quotes with no intervening blanks are counted as a single quote.

Note that if a quote edit descriptor occurs within a character constant and includes an apostrophe, the apostrophe must be represented by two consecutive apostrophes.

14.5.2 H EDITING

The nH edit descriptor causes character information to be written from the n characters (including blanks) following the H of the nH edit descriptor in the format specification itself. An H edit descriptor must not be used on input.

Note that if an H edit descriptor occurs within a character constant and includes an apostrophe, the apostrophe must be represented by two consecutive apostrophes which are counted as one character in specifying n.

14.5.3 POSITIONAL EDITING.

The T, TL, TR, and X edit descriptors specify the position at which the next character will be transmitted to or from the record.

The position specified by a T edit descriptor may be in either direction from the current position. On input, this allows portions of a record to be processed more than once, possibly with different editing.

The position specified by an X edit descriptor is forward from the current position. On input, a position beyond the last character of the record may be specified if no characters are transmitted from such positions.

On output, a T, TL, TR, or X edit descriptor does not by itself cause characters to be transmitted and therefore does not by itself affect the length of the record. If characters are transmitted to positions at or after the position specified by a T, TL, TR, or X edit descriptor, positions skipped and not previously filled are filled with blanks. The result is as if the entire record were intially filled with blanks. i

On output, a character in the record may be replaced. However, a T, TL, TR, or X edit descriptor never directly causes a character already placed in the record to be replaced. Such edit descriptors may result in positioning so that subsequent editing causes a replacement.

14.5.3.1 T, TL, and TR Editing.

The T$\underline{c}$ edit descriptor indicates that the transmission of the next character to or from a record is to occur at the $\underline{c}$th character position.

The TL$\underline{c}$ edit descriptor indicates that the transmission of the next character to or from the record is to occur at the character position $\underline{c}$ characters backward from the current position. However, if the current position is less than or equal to position $\underline{c}$, the TL$\underline{c}$ edit descriptor indicates that the transmission of the next character to or from the record is to occur at position one of the current record.

The TR$\underline{c}$ edit descriptor indicates that the transmission of the next character to or from the record is to occur at the character position $\underline{c}$ characters forward from the current position.

14.5.3.2 X Editing.

The $\underline{n}$X edit descriptor indicates that the transmission of the next character to or from a record is to occur at the position $\underline{n}$ characters forward from the current position.

14.5.4 SLASH EDITING.

The slash edit descriptor indicates the end of data transfer on the current record.

On input from a file connected for sequential access, the remaining portion of the current record is skipped and the file is positioned at the beginning of the next record. This record becomes the current record. On output to a file connected for sequential access, a new record is created and becomes the last and current record of the file.

Note that a record that contains no characters may be written on

output. If the file is an internal file or a file connected for direct access, the record is filled with blank characters. Note also that an entire record may be skipped on input.

For a file connected for direct access, the record number is increased by one and the file is positioned at the beginning of the record that has that record number. This record becomes the current record.

14.5.5 COLON EDITING.

The colon edit descriptor terminates format control if there are no more items in the input/output list (14.3). The colon edit descriptor has no effect if there are more items in the input/output list.

14.5.6 S, SP, AND SS EDITING.

The S, SP, and SS edit descriptors may be used to control optional plus characters in numeric output fields. At the beginning of execution of each formatted output statement, the processor will not produce a plus in numeric output fields. If an SP edit descriptor is encountered in a format specification, the processor must produce a plus in any subsequent position that normally would lack the plus. If an SS edit descriptor is encountered, the processor must not produce a plus in any subsequent position that normally contains an optional plus. If an S edit descriptor is encountered, the procesor reverts to not producing a plus in numeric output fields.

The S, SP, and SS edit descriptors affect only I, F, E, D, and G editing during the execution of an output statement. The S, SP, and SS edit descriptors have no effect during the execution of an input statement.

14.5.7 P EDITING.

A scale factor is specified by a P edit descriptor, which is of the form:

    kP

where k is an optionally signed integer constant, called the scale factor.

14.5.7.1 Scale Factor.

The value of the scale factor is zero at the beginning of execution of each input/output statement. It applies to all subsequently interpreted F, E, D, and G edit descriptors until another scale factor is encountered, and then that scale factor is established. Note that reversion of format control (14.3) does not affect the

established scale factor.

The scale factor k affects the appropriate editing in the following manner:

(1) On input, with F, E, D, and G editing (provided that no exponent exists in the field) and F output editing, the scale factor effect is that the externally represented number equals the internally represented number multiplied by 10**k.

(2) On input, with F, E, D, and G editing, the scale factor has no effect if there is an exponent in the field.

(3) On output, with E and D editing, the basic real constant (4.4.1) part of the quantity to be produced is multiplied by 10**k and the exponent is reduced by k.

(4) On output, with G editing, the effect of the scale factor is suspended unless the magnitude of the datum to be edited is outside the range that permits the use of F editing. If the use of E editing is required, the scale factor has the same effect as with E output editing.

14.5.8 BN_AND_BZ_EDITING.

The BN and BZ edit descriptors may be used to specify the interpretation of blanks, other than leading blanks, in numeric input fields. At the beginning of execution of each formatted input statement, such blank characters are interpreted as zeros or are ignored depending on the value of the BLANK= specifier (13.10.1) currently in effect for the unit. If a BN edit descriptor is encountered in a format specification, all such blank characters in succeeding numeric input fields are ignored. The effect of ignoring blanks is to treat the input field as if blanks had been removed, the remaining portion of the field right justified, and the blanks replaced as leading blanks. However, a field of all blanks has the value zero. If a BZ edit descriptor is encountered in a format specification, all such blank characters in succeeding numeric input fields are treated as zeros.

The BN and BZ edit descriptors affect only I, F, E, D, G, O, and Z editing during execution of an input statement. They have no effect during execution of an output statement.

14.5.9 NUMERIC_EDITING

The I, F, E, D, and G edit descriptors are used to specify input/output of integer, real, double precision, half precision, and complex data. The following general rules apply:

(1) On input, leading blanks are not significant. The

interpretation of blanks, other than leading blanks, is determined by a combination of any BLANK= specifier and any BN or BZ blank control that is currently in effect for the unit (14.5.8). Plus signs may be omitted. A field of all blanks is considered to be zero.

(2) On input, with F, E, D, and G editing, a decimal point appearing in the input field overrides the portion of an edit descriptor that specifies the decimal point location. The input field may have more digits than the processor uses to approximate the value of the datum.

(3) On output, the representation of a positive or zero internal value in the field must not be prefixed with a plus unless a SP edit descriptor is in effect. The representation of a negative internal value in the field must be prefixed with a minus. However, the processor must not produce a negative signed zero in a formatted output record.

(4) On output, the representation is right justified in the field. If the number of characters produced by the editing is smaller than the field width, leading blanks will be inserted in the field.

(5) On output, if the number of characters produced exceeds the field width or if an exponent exceeds its specified length using the E*w*.*d*E*e* or G*w*.*d*E*e* edit descriptor, 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. Note that when an SP edit descriptor is in effect, a plus is not optional (14.5.6).

14.5.9.1 Integer Editing

The I*w* and I*w*.*m* edit descriptors indicate that the field to be edited occupies *w* positions. The specified input/output list item must be of type integer. On input, the specified list item will become defined with an integer datum. On output, the specified list item must be defined with an integer datum.

On input, an I*w*.*m* edit descriptor is treated identically to an I*w* edit descriptor.

In the input field, the character string must be in the form of an optionally signed integer constant, except for the interpretation of blanks (14.5.9, item (1)).

The output field for the I*w* edit descriptor consists of zero or more leading blanks 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 in the form of an unsigned integer constant without leading zeros. Note that an integer constant always consists of at least one digit.

The output field for the I*w*.*m* edit descriptor is the same as for the I*w* edit descriptor, except that the unsigned integer constant consists of at least *m* digits and, if necessary, has leading zeros. The value of *m* must not exceed the value of *w*. If *m* is zero and the value of the internal datum is zero, the output field consists of only blank characters, regardless of the sign control in effect.

14.5.9.2 Real, Double - , and Half - Precision Editing

The F, E, D, and G edit descriptors specify the editing of real, double precision, half precision, and complex data. An input/output list item corresponding to an F, E, D, or G edit descriptor must be real, double precision, half precision, or complex. An input list item will become defined with a datum whose type is the same as that of the list item. An output list item must be defined with a datum whose type is the same as that of the list item.

14.5.9.2.1 F EDITING

The F*w*.*d* edit descriptor indicates that the field occupies *w* positions, the fractional part of which consists of *d* digits.

The input field consists of an optional sign, followed by a string of digits optionally containing a decimal point. If the decimal point is omitted, the rightmost *d* digits of the string, with leading zeros assumed if necessary, are interpreted as the fractional part of the value represented. The string of digits may contain more digits than a processor uses to approximate the value of the constant. The basic form may be followed by an exponent of one of the following forms:

(1) Signed integer constant

(2) E followed by zero or more blanks, followed by an optionally signed integer constant

(3) D followed by zero or more blanks, followed by an optionally signed integer constant

(4) S followed by zero or more blanks, followed by an optionally signed integer constant.

An exponent containing a D or an S is processed identically to an exponent containing an E.

The 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 that contains a decimal point and represents the magnitude of the internal value, as modified by the established scale factor and rounded to d fractional digits. Leading zeros are not permitted except for an optional zero immediately to the left of the decimal point if the magnitude of the value in the output field is less than one. The optional zero must appear if there would otherwise be no digits in the output field.

14.5.9.2.2 E AND D EDITING

The Ew.d, Dw.d, and Ew.dEe edit descriptors indicate that the external field occupies w positions, the fractional part of which consists of d digits, unless a scale factor greater than one is in effect, and the exponent part consists of e digits. The e has no effect on input.

The form of the input field is the same as for F editing (14.5.9.2.1).

The form of the output field for a scale factor of zero is:

    [±] [0] . x1x2...xd exp

where: ± signifies a plus or a minus (14.5.9)

x1x2...xd are the d most significant digits of the value of the datum after rounding

exp is a decimal exponent, of one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
Ew.d	¦exp¦≤99	E±z1z2
	99<¦exp¦≤999	±z1z2z3
Ew.dEe	¦exp¦≤(10**e)-1	E±z1z2...ze
Dw.d	¦exp¦≤99	D±z1z2
	99<¦exp¦≤999	±z1z2z3

where z is a digit. The sign in the exponent is required. A plus sign must be used if the exponent value is zero. The forms Ew.d and Dw.d must not be used if ¦exp¦ > 999.

The scale factor k controls the decimal normalization (14.5.7). If -d < k ≤ 0, the output field contains exactly ¦k¦ leading zeros and d - ¦k¦ significant digits after the decimal point. If 0 < k < d +

2, the output field contains exactly k significant digits to the left of the decimal point and d - k + 1 significant digits to the right of the decimal point. Other values of k are not permitted.

14.5.9.2.3 G EDITING The Gw.d and Gw.dEe edit descriptors indicate that the external field occupies w positions, the fractional part of which consists of d digits, unless a scale factor greater than one is in effect, and the exponent part consists of e digits.

G input editing is the same as for F editing (14.5.9.2.1).

The method of representation in the output field depends on the magnitude of the datum being edited. Let N be the magnitude of the internal datum. If N < 0.1 or N ≥ 10**d, Gw.d output editing is the same as kPEw.d output editing and Gw.dEe output editing is the same as kPEw.dEe output editing, where k is the scale factor currently in effect. If N is greater than or equal to 0.1 and is less than 10**d, the scale factor has no effect, and the value of N determines the editing as follows:

Magnitude of Datum	Equivalent Conversion
0.1≤N<1	F(w-n).d, n('b')
1≤N<10	F(w-n).(d-1), n('b')
.	.
.	.
.	.
10**(d-2)≤N<10**(d-1)	F(w-n).1, n('b')
10**(d-1)≤N<10**d	F(w-n).0, n('b')

where: b is a blank

n is 4 for Gw.d and e+2 for Gw.dEe

Note that the scale factor has no effect unless the magnitude of the datum to be edited is outside of the range that permits effective use of F editing.

14.5.9.2.4 COMPLEX EDITING

A complex datum consists of a pair of separate real data; therefore, the editing is specified by two successively interpreted F, E, D, or G edit descriptors. The first of the edit descriptors specifies the real part; the second specifies the imaginary part. The two edit descriptors may be different. Note that nonrepeatable edit descriptors may appear between the two successive F, E, D, or G edit

descriptors.

14.5.10 L EDITING

The Lw edit descriptor indicates that the field occupies w positions. The specified input/output list item must be of type logical. On input, the list item will become defined with a logical datum. On output, the specified list item must be defined with a logical datum.

The input field consists of optional blanks, optionally followed by a decimal point, followed by a T for true or F for false. The T or F may be followed by additional characters in the field. Note that the logical constants .TRUE. and .FALSE. are acceptable input forms.

The 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.

14.5.11 A EDITING

The A[w] edit descriptor may be used with an input/output list item of type character (14.5.11.1), and the Aw edit descriptor may be used with an input/output list item of any noncharacter type except bit (14.5.11.2).

14.5.11.1 A Editing of Character Data

If a field width w is specified with the A edit descriptor, the field consists of w characters. If a field width w is not specified with the A edit descriptor, the number of characters in the field is the length of the character input/output list item.

Let len be the length of the input/output 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 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.

If the specified field width w for A output is greater than len, the output field will consist of w-len blanks followed by the len characters from the internal representation. If the specified field width w is less than or equal to len, the output field will consist of the leftmost w characters from the internal representation.

14.5.11.2 A Editing of Noncharacter Data

When used with an input/output list item of noncharacter type, the Aw edit descriptor indicates character code conversion (Appendix A)

of w characters, with left justification and trailing blank fill. The field width specifier w must be a positive nonzero integer for this usage. In the following, a is the maximum number of characters that can be stored in the datum at one time, and b is the number of bits in a single character storage unit.

On input, if w is less than or equal to a, the next w characters from the record are converted to w*b bits of character code, and are suffixed with (a-w) character code blanks. The result is transmitted to the noncharacter datum corresponding to the next input/output list item. If w is greater than a, (w-a) characters from the input record are skipped; the next a characters are converted and transmitted to the datum.

On output, if w is less than or equal to a, the leftmost (most significant) w*b bits of the list item datum are converted and transmitted to the output record. If w is greater than a, w-a leading blanks are transmitted, followed by a characters converted from the datum.

14.5.12 PROCESSOR-DEPENDENT EDITING

The R, O and Z edit descriptors are used, and the A edit descriptor may be used, to specify processor-dependent editing for the physical contents of numeric storage units. The editing consists of direct octal, hexadecimal or character code conversion between internal storage and character strings of a record. Conversion proceeds on a bit-by-bit basis; no numeric or logical significance is attached to the data. Any data type except types bit and character may be edited with the R edit descriptor. Any data type, except bit, may be edited with the A, O, and Z edit descriptors.

The number of bits corresponding to an external character varies with the edit descriptor.

14.5.12.1 R Editing

The Rw edit descriptor indicates character code conversion (Appendix A) of w characters, with right justification and leading zero bit fill. In the following, a is the maximum number of characters that can be stored in the datum at one time, and b is the number of bits in a single character storage unit.

On input, if w is less than or equal to a, (a-w)*b leading zero fill bits are transmitted to the leftmost (most significant) bit positions of the list item datum, followed by the next w characters of the input record converted to character code. If w is greater

than _a_, (_w_-_a_) input characters are skipped; the next _a_ characters are converted to character code and transmitted to the datum.

On output, if _w_ is less than or equal to _a_, the rightmost (least significant) _w_*_b_ bits of the list item storage unit are extracted and, beginning with the leftmost extracted bit, converted to _w_ characters and transmitted to the output record. If _w_ is greater than _a_, _w_-_a_ leading blank characters are transmitted, followed by _a_ characters converted from the storage unit.

14.5.12.2 O Editing

The O_w_ and O_w_._m_ edit descriptors indicate octal conversion of _w_ characters. _m_ is significant only on output. Each character corresponds to 3 physical bits. The octal digits and bit values are:

Octal Digit	Bit Value
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

On input, the input field consists of zero or more leading blanks followed by a string of octal digits. Optionally the string may contain embedded blanks; each blank is significant and is equivalent to an octal digit zero. The string may contain a maximum of _o_ characters including embedded blanks, but excluding leading blanks. _o_ = (_b_+2)/3 where _b_ is the number of bits in the list item datum. If _o_*3 is greater than _b_ then the leftmost (_o_*3-_b_) bits corresponding to the input field must be zero. For a string of _n_ characters, (_o_-_n_)*3-2 leading zero fill bits are transmitted to the leftmost (most significant) bit positions of the list item storage unit, followed by the _n_ characters of the input record converted to _n_*3 bits.

If the input field contains no characters except the character blank, the field is interpreted as equivalent to zero (0).

On output, if w is less than or equal to b/3, the rightmost w*3 bits of the list item storage unit are converted to octal digit characters and transmitted to the output record. If w is greater than o, w-o leading blank characters are transmitted, followed by o octal digits converted from the storage unit. If w is equal to o, all the bits of the list item datum are converted to octal digit characters and transmitted.

If m is specified on output, the value of m must not exceed the value of w and the following additional editing applies:

> A maximum of w-m leading octal zero characters are replaced with blank characters, proceeding from left to right. A minimum of m octal digit characters are not replaced.

If the number of octal digit characters produced exceeds the field width w, the entire field is filled with asterisks.

14.5.12.3 Z Editing

The Zw and Zw.m edit descriptors indicate hexadecimal conversion of w characters. m is significant only on output. Each character corresponds to 4 physical bits. The hexadecimal digits and bit values are:

Hexadecimal Digit	Bit Value
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

On input, the input field consists of zero or more leading blanks

followed by a string of hexadecimal digits. Optionally the string may contain embedded blanks; each blank is significant and is equivalent to an hexadecimal digit zero. The string may contain a maximum of z characters including embedded blanks, but excluding leading blanks. z = (b+3)/4 where b is the number of bits in the list item datum. If z*4 is greater than b then the leftmost (z*4-b) bits corresponding to the input field must be zero. For a string of n characters, (z-n)*4 leading zero fill bits are transmitted to the leftmost (most significant) bit positions of the list item datum, followed by the n characters of the input record converted to n*4 bits.

If the input field contains no characters except the character blank, the field is interpreted as equivalent to zero (0).

On output, if w is less than or equal to b/4, the rightmost w*3 bits of the list item storage unit are converted to hexadecimal digit characters and transmitted to the output record. If w is greater than z, w-z leading blank characters are transmitted, followed by z hexadecimal digits converted from the storage unit. If w is equal to z, all the bits of the list item storage unit are converted to hexadecimal digit characters and transmitted.

If m is specified on output, the value of m must not exceed the value of w and the following additional editing applies:

> A maximum of w-m leading hexadecimal zero characters are replaced with blank characters, proceeding from left to right. A minimum of m hexadecimal digit characters are not replaced.

If the number of hexadecimal digit characters produced exceeds the field width w, the entire field is filled with asterisks.

14.5.13 B EDITING

The Bw edit descriptor formats bit data during input/output operations.

On input, the value in the input field is converted to type bit and is assigned to the input list item. One bit is input. The input list item must be of type bit.

The data that appears in a B input field must be in the proper format. The B input field must contain a 0 or 1 in the rightmost column; all other columns must be blank.

On output, the value of the output list item is converted to a string of characters. These characters are placed in the output field. One bit is output. The output list item must be of type bit.

The B output field contains a 0 or a 1 if the value of the output list item is B"0" or B"1", respectively. All other columns are blank.

14.6 LIST-DIRECTED FORMATTING

The characters in one or more list-directed records constitute a sequence of values and value separators. The end of a record has the same effect as a blank character, unless it is within a character constant. Any sequence of two or more consecutive blanks is treated as a single blank, unless it is within a character constant.

Each value is either a constant, a null value, or of 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 r successive null values. Neither of these forms may contain embedded blanks, except where permitted within the constant c.

A value separator is one of the following:

(1) A comma optionally preceded by one or more contiguous blanks and optionally followed by one or more contiguous blanks

(2) A slash optionally preceded by one or more contiguous blanks and optionally followed by one or more contiguous blanks

(3) One or more contiguous blanks between two constants or following the last constant

14.6.1 LIST-DIRECTED INPUT

Input forms acceptable to format specifications for a given type are acceptable for list-directed formatting, except as noted below. The form of the input value must be acceptable for the type of the input list item. Blanks are never used as zeros, and embedded blanks are not permitted in constants, except within character constants and complex constants as specified below. Note that the end of a record has the effect of a blank, except when it appears within a character constant. A Boolean constant may be used as an input form only if the corresponding input list item is of type Boolean.

When the corresponding input list item is of type real, double precision, or half precision, the input form is that of a numeric

input field. A _numeric input field_ is a field suitable for F editing (14.5.9.2) which is assumed to have no fractional digits unless a decimal point appears within the field.

When the corresponding list item is of type complex, the input form consists of a left parenthesis followed by an ordered pair of numeric input fields separated by a comma, and followed by a right parenthesis. The first numeric input field is the real part of the complex constant and the second is the imaginary part. Each of the numeric input fields may be preceded or followed by blanks. The end of a record may occur between the real part and the comma or between the comma and the imaginary part.

When the corresponding list item is of type bit, the input form is the same as a B input field.

When the corresponding input list item is of type Boolean, the input form is one of the following:

(1) an octal constant (4.9.1.2),

(2) a hexadecimal constant (4.9.1.3),

(3) a Hollerith constant of the form "_f_" (4.9.1.1),

(4) an integer input field, or

(5) a numeric input field.

When the corresponding list item is of type logical, the input form must not include either slashes or commas among the optional characters permitted for L editing (14.5.10).

When the corresponding list item is of type character, the input form consists of a nonempty string of characters enclosed in apostrophes. Each apostrophe within a character constant must be represented by two consecutive apostrophes without an intervening blank or end of record. Character constants may be continued from the end of one record to the beginning of the next record. The end of the record does not cause a blank or any other character to become part of the constant. The constant may be continued on as many records as needed. The characters blank, comma, and slash may appear in character constants.

Let _len_ be the length of the list item, and let _w_ be the length of the character constant. If _len_ is less than or equal to _w_, the leftmost _len_ characters of the constant are transmitted to the list item. If _len_ is greater than _w_, the constant is transmitted to the leftmost _w_ characters of the list item and the remaining _len-w_ characters of the list item are filled with blanks. Note that the

effect is as though the constant were assigned to the list item in a character assignment statement (10.4).

A null value is specified by having no characters between successive value separators, no characters preceding the first value separator in the first record read by each execution of a list-directed input statement, or the r* form. A null value has no effect on the definition status of the corresponding input list item. If the input 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 a single null value may represent an entire complex constant. Note that the end of a record following any other separator, with or without separating blanks, does not specify a null value.

A slash encountered as a value separator during execution of a list-directed input statement causes termination of execution of that input statement after the assignment of the previous value. If there are additional items in the input list, the effect is as if null values had been supplied for them.

Note that all blanks in a list-directed input record are considered to be part of some value separator except for the following:

(1) Blanks embedded in a character constant

(2) Embedded blanks surrounding the real or imaginary part of a complex constant

(3) Leading blanks in the first record read by each execution of a list-directed input statement, unless immediately followed by a slash or comma

14.6.2 LIST-DIRECTED OUTPUT

The form of the values produced is the same as that required for input, except as noted otherwise. With the exception of character constants, the values are separated by one of the following:

(1) One or more blanks

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

The processor may begin new records as necessary, but, except for complex constants and character constants, the end of a record must not occur within a constant and blanks must not appear within a constant.

Bit output constants are produced with the effect of a B edit descriptor.

Boolean output constants are produced in the form Z"z[z]...", where z is a hexadecimal digit. Leading zeros are suppressed.

Logical output constants are T for the value true and F for the value false.

Integer output constants are produced with the effect of an Iw edit descriptor, for some reasonable value of w. The value chosen for w must be large enough to hold all of the significant digits of the integer as well as the sign if it is negative.

Real, double precision, and half precision constants are produced with the effect of either an F edit descriptor or an E edit descriptor, depending on the magnitude x of the value and a range 10**d1 ≤ x < 10**d2, where d1 and d2 are -6 and +9, respectively. If the magnitude x is within this range, the constant is produced using 0PFw.d; otherwise, 1PEw.dEe is used. Reasonable processor-dependent values of w, d, and e are used for each of the cases involved.

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

Character constants produced 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 record that begins with the continuation of a character constant from the preceding record.

If two or more successive values in an output record produced 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.

Slashes, as value separators, and null values are not produced by list-directed formatting.

Each output record begins with a blank character to provide carriage control when the record is printed.

14.7 NAMELIST FORMATTING

The form of a NAMELIST block is one of the following:

$*grpname* *namval*[,*namval*]...[,]$[END]

&*grpname* *namval*[,*namval*]...[,]&[END]

$*grpname* *namval*[,*namval*]...[,]&[END]

&*grpname* *namval*[,*namval*]...[,]$[END]

where: *grpname* is the group name of the block

namval is one of the forms:

vname = *c*

aname[(*s*)]=[*r**]*c*[,[*r**]*c*]...

vname(*i1*:*i2*) = *c*

aname(*s*)(*i1*:*i2*) = *c*

where: *vname* is a variable name

c is a constant

aname is an array name

s is an array subscript in which each subscript expression is an integer constant. The number of subscript expressions in *s* must be equal to the number of dimensions in the array declarator for the array name.

r is an unsigned, nonzero, integer constant

i1 and *i2* are integer constants

(*i1*:*i2*) designates a character substring of a *vname* or a *aname* of type character

The optional form *r***c* is equivalent to *r* successive appearances of the constant *c*.

A NAMELIST block consists of one or more formatted records. The initial record of the block may contain only $*grpname*, &*grpname*, or one of these followed by one or more occurrences of *namval* forms. The initial and subsequent records of a NAMELIST block may end with a comma that occurs after a constant *c* or a comma that occurs after the real part of a complex constant that is continued on the next record. The initial or subsequent records of a NAMELIST block may end with a constant *c* provided that the record immediately precedes a record containing only $, $END, & or &END. The final record of a

NAMELIST block must end with $, $END, & or &END. Note that the final record may be the initial record. For example,

```
&GROUP   V=3.7, &END
```

In each record of a NAMELIST block, column one is reserved for carriage control. On input, the character in column one is ignored. On output, a carriage control character is placed in column one of each record. A NAMELIST block is written in columns two through the end of each input record.

The case of alphabetic letters in _grpname_, _vname_, _aname_, $END, and &END is not significant in identifying the entity.

A blank must not occur within the strings:

(1) $_grpname_
(2) &_grpname_
(3) _vname_
(4) _aname_[(_s_)]
(5) _vname_(_i1_:_i2_)
(6) _aname_(_s_)(_i1_:_i2_)
(7) $END
(8) &END

Forms (1) and (2) must be followed by an end of record or one or more blanks before forms (2) through (8) may appear. Forms (7) and (8) must be separated from the other forms by a comma or one or more blanks or by the end of a record.

14.7.1 NAMELIST INPUT

The group name of the NAMELIST block being transmitted must appear in the READ statement being executed. Each variable name and array name in the block must appear in the NAMELIST group referred to by the READ statement.

Each constant _c_ must agree with the type of the corresponding list item as follows:

(1) A logical, character, bit, or complex constant must be of the same type as the corresponding input list item. A character constant is truncated from the right or extended to the right with blank characters, if necessary, to yield a character constant of the same length as the corresponding character variable, array element, or substring.

(2) An integer, real, half precision, or double precision constant may be used for an integer, real, half precision, or double precision input list item. The constant is

converted to the type of the list item during transmission. For conversion to real, half precision, or double precision, an integer constant has an implied decimal point to the right of the rightmost digit.

The number of constants specified for a list item must not exceed the size of the list item.

The form aname(s)=[r*]c[,[r*]c]... causes the first constant to be assigned to aname (s). Each subsequent constant is assigned to the array element with a subscript value (Table 1) one greater than the previous element assigned to. The size of aname must not be exceeded.

The forms of a logical constant having the value true are:

T
.T.
.TRUE.

The forms of a logical constant having the value false are:

F
.F.
.FALSE.

A character constant must have the same form as if it appeared in a statement of the executable program; i.e., the delimiting apostrophes must be present. If a character constant occupies more than one record, each continuation of the character constant must extend to the end of each record preceding a continuation record. A character constant continued across a record begins in column two of the continuation record.

The forms of integer, real, double precision, half precision, and complex constants are as described for list directed input (14.6.1) except that embedded blanks are allowed.

A Boolean constant must be an octal constant (4.9.1.2), a hexadecimal constant (4.9.1.3), or a Hollerith constant of the form "f" (4.9.1.1).

The character blank is significant only in a character constant, or Boolean constant of the form "f". The BLANK= specifier has no effect on NAMELIST editing. A c form must contain one or more nonblank characters and be a valid constant or an error condition exists. Note that a c of type character will always contain at least two nonblank characters due to the apostrophes surrounding the character constant.

14.7.2 NAMELIST OUTPUT

On output, each NAMELIST block is terminated with the characters &END.

The processor begins a new record for each block transferred. Column one of the first record of each block contains a carriage control character followed by an & followed by the group name.

The processor begins a new record for the block terminator &END. Column one of the block terminator record contains a carriage control followed immediately by &END.

The processor may begin new records as necessary, but, except for complex constants and character constants, the end of a record must not occur within a constant. A new record is begun with the carriage control character blank in column one, and the leftmost character of the constant in column two.

Logical constants are produced as T for the value true and F for the value false.

Character constants are produced with delimiting apostrophes.

Boolean constants are produced in the form Z"z[z]...", where z is a hexadecimal digit. Leading zeros are suppressed.

Integer constants are produced with the effect of an Iw edit descriptor, for some reasonable value of w. The value chosen for w must be large enough to hold all of the significant digits of the integer as well as the sign if it is negative.

Except for the value zero, real, double precision, and half precision constants are produced with the effect of an E edit descriptor with scale factor of zero. No digits are produced before the decimal point. The number of digits produced to the right of the decimal point is the minimum number necessary to accomodate representation of the internal datum. Trailing zeros are eliminated. The string 0.0 is produced for the value zero.

Complex constants are produced as a pair of real constants enclosed in parentheses and with a comma separating the real and imaginary parts. Each real constant is produced as described in the preceding paragraph.

Bit constants are produced with the effect of a Bw edit descriptor.

15.0 MAIN PROGRAM

A main program is a program unit that does not have a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement. It may have a PROGRAM statement as its first statement.

There must be exactly one main program in an executable program. Execution of an executable program begins with the execution of the first executable statement of the main program.

15.1 PROGRAM STATEMENT

The form of a PROGRAM statement is:

PROGRAM pgm

where pgm is the symbolic name of the main program in which the PROGRAM statement appears.

A PROGRAM statement is not required to appear in an executable program. If it does appear, it must be the first statement of the main program. If omitted, the symbolic name pgm of the main program is the six character name START#.

The symbolic name pgm is global (19.1.1) to the executable program and must not be the same as the name of an external procedure, block data subprogram, or common block in the same executable program. The name pgm must not be the same as any local name in the main program.

15.2 MAIN PROGRAM RESTRICTIONS

The PROGRAM statement may appear only as the first statement of a main program. A main program may contain any other statement except a BLOCK DATA, FUNCTION, SUBROUTINE, or ENTRY statement. The appearance of a SAVE statement in a main program has no effect.

A main program may not be referenced from a subprogram or from itself.

16.0 FUNCTIONS AND SUBROUTINES

16.1 CATEGORIES OF FUNCTIONS AND SUBROUTINES

16.1.1 PROCEDURES.

Functions and subroutines are procedures. There are four categories of procedures:

(1) Intrinsic functions

(2) Statement functions

(3) External functions

(4) Subroutines

Intrinsic functions, statement functions, and external functions are referred to collectively as functions.

External functions and subroutines are referred to collectively as external procedures.

16.1.2 EXTERNAL FUNCTIONS.

There are two categories of external functions:

(1) External functions specified in function subprograms

(2) External functions specified by means other than FORTRAN subprograms

16.1.3 SUBROUTINES.

There are two categories of subroutines:

(1) Subroutines specified in subroutine subprograms

(2) Subroutines specified by means other than FORTRAN subprograms

16.1.4 DUMMY PROCEDURE.

A dummy procedure is a dummy argument that is identified as a procedure (19.2.11).

16.2 REFERENCING A FUNCTION

A function is referenced in an expression and supplies a value to the expression. The value supplied is the value of the function.

16.0 FUNCTIONS AND SUBROUTINES
16.2 REFERENCING A FUNCTION

An intrinsic function may be referenced in the main program or in any procedure subprogram of an executable program.

A statement function may be referenced only in the program unit in which the statement function statement appears.

An external function specified by a function subprogram may be referenced within any other procedure subprogram or the main program of the executable program. A subprogram must not reference itself, either directly or indirectly.

An external function specified by means other than a subprogram may be referenced within any procedure subprogram or the main program of the executable program.

If a character function is referenced in a program unit, the function length specified in the program unit must be an extended integer constant expression.

16.2.1 FORM_OF_A_FUNCTION_REFERENCE.

A function reference is used to reference an intrinsic function, statement function, or external function.

The form of a function reference is:

```
fun ( [a [,a]...] )
```

where: fun is the symbolic name of a function or a dummy procedure

a is an actual argument

The type of the result of a statement function or external function reference is the same as the type of the function name. The type is specified in the same manner as for variables and arrays (4.1.2). The type of the result of an intrinsic function is specified in Table 5 (16.10).

16.2.2 EXECUTION_OF_A_FUNCTION_REFERENCE.

A function reference may appear only as a primary in an arithmetic, logical, or character expression. Execution of a function reference in an expression causes the evaluation of the function identified by fun.

Return of control from a referenced function completes execution of the function reference. The value of the function is available to the referencing expression.

16.3 INTRINSIC FUNCTIONS

Intrinsic functions are supplied by the processor and have a special meaning. The specific names that identify the intrinsic functions, their generic names, function definitions, type of arguments, and type of results appear in Table 5.

An IMPLICIT statement does not change the type of an intrinsic function.

An intrinsic function is _elemental_ if the result can be determined on an element by element basis when applied to an array valued argument. An intrinsic function is _array-valued_ if it appears in the array-valued section of Table 5. An array-valued function is generally one which cannot be evaluated independently for each array element. The value of an array-valued function need not have the same shape as any of its arguments.

None of the array-valued intrinsic functions may be passed as an argument to an external procedure.

16.3.1 SPECIFIC NAMES AND GENERIC NAMES

Generic names simplify the referencing of intrinsic functions, because the same function name may be used with more than one type of argument. Only a specific intrinsic function name may be used as an actual argument when the argument is an intrinsic function.

If a generic name is used to reference an intrinsic function, the type of the result (except for intrinsic functions performing type conversion, nearest integer, and absolute value with a complex argument) is the same as the type of the argument. If there is only one argument and it is of type Boolean, it is converted to type integer if an argument of type integer is allowed; otherwise it is converted to type real. The type of the result is then the type to which the Boolean argument was converted.

For those intrinsic functions that have more than one argument, all arguments must be of the same type with the exception that a Boolean argument is allowed in the same intrinsic function reference as an arithmetic argument. In this case each Boolean argument is converted to the type of the arithmetic argument or arguments. If all the arguments are of type Boolean, each of them is converted to type integer.

If the specific name or generic name of an intrinsic function appears in the dummy argument list of a function or subroutine in a subprogram, that symbolic name does not identify an intrinsic function in the program unit. The data type identified with the symbolic name is specified in the same manner as for variables and arrays (4.1.2).

A name in an INTRINSIC statement must be the specific name or generic name of an intrinsic function.

16.3.2 REFERENCING AN INTRINSIC FUNCTION.

An intrinsic function is referenced by using its reference as a primary in an expression. For each intrinsic function described in Table 5, execution of an intrinsic function reference causes the actions specified in Table 5, and the result depends on the values of the actual arguments. The resulting value is available to the expression that contains the function reference.

The actual arguments that constitute the argument list must agree in order and number with the specification in Table 5. They must also agree in type except that an actual argument of type Boolean may appear whenever Table 5 permits an argument of arithmetic type. An actual argument may be any expression of an allowed type. An actual argument in an intrinsic function reference may be any expression except a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses unless the operand is the symbolic name of a constant.

A specific name of an intrinsic function that appears in an INTRINSIC statement may be used as an actual argument in an external procedure reference; however, the names of intrinsic functions for type conversion, lexical relationship, and for choosing the largest or smallest value must not be used as actual arguments. Note that such an appearance does not cause the intrinsic function to be classified as an external function (19.2.10).

16.3.3 INTRINSIC FUNCTION ARGUMENTS AND RESULTS

If a generic name or a specific name is used to reference an elemental intrinsic function, the shape of the result is the same as the shape of the argument with the greatest number of dimensions. If the arguments are all scalar, the result is scalar. If a generic name or a specific name is used to reference an array-valued intrinsic function, the shape of the result is specified by Table 5.

For those elemental intrinsic functions that have more than one argument, all arguments must be conformable.

The actual arguments that constitute the argument list must agree in order, number, and type with the specification in Table 5. For elemental intrinsic functions, the actual arguments may be any expression including an array expression. For array-valued intrinsic functions, a scalar actual argument may not be used where Table 5 indicates an array is expected.

16.3.4 INTRINSIC FUNCTION RESTRICTIONS.

Arguments for which the result is not mathematically defined or exceeds the numeric range of the processor cause the result of the function to become undefined.

Restrictions on the range of arguments and results for intrinsic functions are described in 16.10.1.

16.3.5 ARRAY REDUCTION INTRINSIC FUNCTIONS

16.3.5.1 Array Reduction to a Scalar

An array reduction function can be used to reduce an array to a scalar value of the same type or to reduce an array along a specific dimension.

```
name(A)
name(A,[DIM=]d)
name(A,[DIM=]d,[MASK=]m)
name(A,MASK=m)
```

where:

- A is an array, array section, or array expression with n dimensions such that 1 ≤ d ≤ n
- d is a scalar integer value specifying the dimension along which the function is applied (1 ≤ d ≤ n)
- m is a type logical array, array section, array expression, or scalar conformable to a

The function applies to all elements of the array, array section, or array expression A. An example would be SUM(C) which would return the arithmetic sum of all the elements of the array C.

16.3.5.2 Array Reduction Along a Dimension

If the shape of array A is (d1,d2,...,d(i-1),di,d(i+1),...,dn) and an array reduction function is applied along dimension i, the resultant array has shape

```
(d1,d2,...,d(i-1),d(i+1),...,dn)
```

The value of element (s1,s2,...,s(i-1),s(i+1),...,sn) of the result array is given by

```
name(A(s1,s2,...,s(i-1),*,s(i+1),...,sn)[,MASK=m])
```

If d is omitted, the entire array is reduced to a scalar.

A masked reduction applies to all of the elements of A (selected by d) where corresponding values of elements in m are true. A and m must be conformable. For example, SUM(C,MASK=C .GT. 0.0) would form the arithmetic sum of all elements of C whose values are greater than zero. Each masked reduction function has a default value that is returned if all of the elements of m are false.

If m is omitted, the array is reduced as if m was an array conformable with A and all of its elements were true.

16.4 STATEMENT FUNCTION

A statement function is a procedure specified by a single statement that is similar in form to an arithmetic, Boolean, bit, logical, or character assignment statement. A statement function statement must appear only after the specification statements and before the first executable statement of the program unit in which it is referenced (3.5).

A statement function statement is classified as a nonexecutable statement; it is not a part of the normal execution sequence.

16.4.1 FORM OF A STATEMENT FUNCTION STATEMENT.

The form of a statement function statement is:

fun ([d [,d]...]) = e

where: fun is the symbolic name of the statement function

d is a statement function dummy argument

e is an expression

The relationship between fun and e must conform to the assignment rules in 10.1, 10.2, and 10.4. Note that the type of the expression may be different from the type of the statement function name.

Each d is a variable name called a statement function dummy argument. The statement function dummy argument list serves only to indicate order, number, and type of arguments for the statement function. The variable names that appear as dummy arguments of a statement function have a scope of that statement (19.1). A given symbolic name may appear only once in any statement function dummy argument list. The symbolic name of a statement function dummy argument may be used to identify other dummy arguments of the same type in different statement function statements. The name may also be used to identify a variable of the same type appearing elsewhere in the program unit, including its appearance as a dummy argument in a FUNCTION, SUBROUTINE, or ENTRY statement. The name must not be used to identify any other entity in the program unit except a

common block.

Each primary of the expression *e* must be one of the following:

(1) A constant

(2) The symbolic name of a constant

(3) A variable reference

(4) An array element reference

(5) An intrinsic function reference

(6) A reference to a statement function for which the statement function statement appears in the same program unit

(7) An external function reference

(8) A dummy procedure reference

(9) An expression enclosed in parentheses that meets all of the requirements specified for the expression *e*

(10) A substring reference

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 as the statement function statement. A substring reference may not have the form *v*([*e*1]:[*e*2]) where *v* is a statement function dummy argument.

If a statement function dummy argument name is the same as the name of another entity, the appearance of that name in the expression of a statement function statement is a reference to the statement function dummy argument. A dummy argument that appears in a FUNCTION or SUBROUTINE statement may be referenced in the expression of a statement function statement within the subprogram. A dummy argument that appears in an ENTRY statement that precedes a statement function statement may be referenced in the expression of the statement function statement within the subprogram.

16.4.2 REFERENCING A STATEMENT FUNCTION.

A statement function is referenced by using its function reference as a primary in an expression.

Execution of a statement function reference results in:

(1) evaluation of actual arguments that are expressions,

(2) conversion, if necessary of the values of the actual arguments to the types of the corresponding dummy arguments according to the rules for assignment, association of resulting values with the corresponding dummy arguments,

(3) evaluation of the expression _e_, and

(4) conversion, if necessary, of an arithmetic expression value to the type of the statement function according to the assignment rules in 10.1 or a change, if necessary, in the length of a character expression value according to the rules in 10.4.

The resulting value is available to the expression that contains the function reference.

The actual arguments, which constitute the argument list, must agree in order and number with the corresponding dummy arguments. An actual argument in a statement function reference may be any expression except a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses unless the operand is the symbolic name of a constant.

When a statement function reference is executed, its actual arguments must be defined.

16.4.3 STATEMENT FUNCTION RESTRICTIONS.

A statement function may be referenced only in the program unit that contains the statement function statement.

The evaluation of the expression _e_ (16.4.1) must not cause the execution of a reference to _fun_ (16.4.1). The symbolic name used to identify a statement function must not appear as a symbolic name in any specification statement except in a type-statement (to specify the type of the function) or as the name of a common block in the same program unit.

An external function reference in the expression of a statement function statement must not cause a dummy argument of the statement function to become undefined or redefined.

The symbolic name of a statement function is a local name (19.1.2) and must not be the same as the name of any other entity in the program unit except the name of a common block.

The symbolic name of a statement function may not be an actual argument. It must not appear in an EXTERNAL statement.

A statement function statement in a function subprogram must not contain a function reference to the name of the function subprogram

or an entry name in the function subprogram.

The length specification of a character statement function or statement function dummy argument of type character must be an extended integer constant expression.

16.5 EXTERNAL FUNCTIONS

An external function is specified externally to the program unit that references it. An external function is a procedure and may be specified in a function subprogram or by some other means.

16.5.1 FUNCTION SUBPROGRAM AND FUNCTION STATEMENT

A function subprogram specifies one or more external functions (16.7). A function subprogram is a program unit that has a FUNCTION statement as its first statement. The form of a function subprogram is as described in 2.4 and 3.5, except as noted in 16.5.3 and 16.7.4.

The form of a FUNCTION statement is:

[typ] FUNCTION fun ([d [,d]...])

where: typ is one of INTEGER, REAL, DOUBLE PRECISION, HALF PRECISION, COMPLEX, LOGICAL, BIT, BOOLEAN, or CHARACTER [*len] where len is the length specification of the result of the character function. len may have any of the forms allowed in a CHARACTER statement (8.4.2). If a length is not specified in a CHARACTER FUNCTION statement, the character function has a length of one.

fun is the symbolic name of the function subprogram in which the FUNCTION statement appears. fun is an external function name.

d is a variable name, array name, or dummy procedure name. d is a dummy argument.

The symbolic name of a scalar-valued function subprogram or an associated entry name of the same type must appear as a variable in the function subprogram. During every execution of the external function, this variable, unless it has length zero, must become defined and, once defined, may be referenced or become redefined. During every execution of a user array-valued function, the array corresponding to fun, unless it has size zero, must become defined and, once defined, may be referenced or become redefined. The value of this variable or array when a RETURN or END statement is executed in the subprogram is the value of the function. If this variable is

a character variable with a length specification that is an asterisk in parentheses, it must not appear as an operand for concatenation except in a character assignment statement (10.4).

An external function in a function subprogram may define one or more of its dummy arguments to return values in addition to the value of the function.

16.5.2 REFERENCING AN EXTERNAL FUNCTION.

An external function is referenced by using its reference as a primary in an expression.

16.5.2.1 Execution of an External Function Reference.

Execution of an external function reference results in:

(1) evaluation of actual arguments that are expressions,

(2) association of actual arguments with the corresponding dummy arguments, and

(3) the actions specified by the referenced function.

The type of the function name in the function reference must be the same as the type of the function name in the referenced function. The length of the character function in a character function reference must be the same as the length of the character function in the referenced function.

When an external function reference is executed, the function must be one of the external functions in the executable program.

16.5.2.2 Actual Arguments for an External Function.

The actual arguments in an external function reference must agree in order and number with the corresponding dummy arguments in the referenced function. They must also agree in type except that an actual argument of type Boolean may have a corresponding dummy argument of type integer or real, and an actual argument of type integer or real may have a corresponding dummy argument of type Boolean. The use of a subroutine name as an actual argument is an exception to the rule requiring agreement of type because subroutine names do not have a type.

An actual argument in an external function reference must be one of the following:

(1) An expression except a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses unless the operand is the symbolic

name of a constant

(2) An array name

(3) An intrinsic function name

(4) An external procedure name

(5) A dummy procedure name

(6) An array expression

(7) An array section name

Note that an actual argument in a function reference may be a dummy argument that appears in a dummy argument list within the subprogram containing the reference.

16.5.3 FUNCTION SUBPROGRAM RESTRICTIONS

A FUNCTION statement must appear only as the first statement of a function subprogram. A function subprogram may contain any other statement except a BLOCK DATA, SUBROUTINE, or PROGRAM statement.

The symbolic name of an external function is a global name (19.1.1) and must not be the same as any other global name, except a common block name, or any local name, except a variable name, in the function subprogram.

Within a function subprogram, the symbolic name of a scalar-valued function specified by a FUNCTION or ENTRY statement must not appear in any other nonexecutable statement, except a type-statement. In an executable statement, such a name may appear only as a variable. The symbolic name of a user array-valued function may appear in an array declarator in any nonexecutable statement that permits an array declarator. The name may appear in a type statement as:

(1) An array declarator provided it is the only appearance as an array declarator in this program unit, or

(2) An array name where the array dimensions are specified in another statement.

If the type of a function is specified in a FUNCTION statement, the function name must not appear in a type-statement. Note that a name must not have its type explicitly specified more than once in a program unit.

If the name of a function subprogram is of type character, each entry name in the function subprogram must be of type character. If

the name of the function subprogram or any entry in the subprogram has a length of (*) declared, all such entities must have a length of (*) declared; otherwise, all such entities must have a length specification of the same integer value.

In a function subprogram, the symbolic name of a dummy argument is local to the program unit and must not appear in an EQUIVALENCE, PARAMETER, SAVE, INTRINSIC, DATA, or COMMON statement, except as a common block name.

A character dummy argument whose length specification is an asterisk in parentheses must not appear as an operand for concatenation, except in a character assignment statement (10.4).

A function specified in a subprogram may be referenced within any other procedure subprogram or the main program of the executable program. A function subprogram must not reference itself, either directly or indirectly.

16.5.4 USER ARRAY-VALUED FUNCTION DECLARATION

An external array-valued function is an external function that returns an array-valued result. An external array-valued function is distinguished from a scalar function by an array declaration for the function name.

16.5.4.1 Shape of a User Array-valued Function Result

The shape of a user array-valued function result is declared in the function subprogram by an array declaration for the function name. The actual shape returned by a reference to the function is determined by evaluation of each of the dimension bound expressions at the time of the reference to the function. Values involved in the determination of the shape may become redefined during the execution of the function but this will not affect the shape.

The shape and type of a user array-valued function must be declared in a procedure information block in the program unit referencing the function and the shape and type must agree with the shape and type specified in the function.

16.5.4.2 User Array-Valued Function Name Usage

The symbolic name of an array-valued function must be declared as an array name in the function subprogram. During every execution of the array-valued function, this array must become defined (but note that an array of size zero is always defined) and once defined, may be referenced or become redefined. The value of this array when a RETURN or END statement is executed in the subprogram is the value of the function.

16.5.4.3 User Array-Valued Function Restrictions

A user array-valued function name may not be passed as an actual argument. A user array-valued function must not contain an ENTRY statement.

16.6 SUBROUTINES

A subroutine is specified externally to the program unit that references it. A subroutine is a procedure and may be specified in a subroutine subprogram or by some other means.

16.6.1 SUBROUTINE SUBPROGRAM AND SUBROUTINE STATEMENT.

A subroutine subprogram specifies one or more subroutines (16.7). A subroutine subprogram is a program unit that has a SUBROUTINE statement as its first statement. The form of a subroutine subprogram is as described in 2.4 and 3.5, except as noted in 16.6.3 and 16.7.4.

The form of a SUBROUTINE statement is:

```
SUBROUTINE sub [( [d [,d]...] )]
```

where: sub is the symbolic name of the subroutine subprogram in which the SUBROUTINE statement appears. sub is a subroutine name.

d is a variable name, array name, or dummy procedure name, or is an asterisk (16.9.3.5). d is a dummy argument.

Note that if there are no dummy arguments, either of the forms sub or sub() may be used in the SUBROUTINE statement. A subroutine that is specified by either form may be referenced by a CALL statement of the form CALL sub or CALL sub().

One or more dummy arguments of a subroutine in a subprogram may become defined or redefined to return results.

16.6.2 SUBROUTINE REFERENCE.

A subroutine is referenced by a CALL statement.

16.6.2.1 Form of a CALL Statement.

The form of a CALL statement is:

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

where: sub is the symbolic name of a subroutine or dummy procedure

_a_ is an actual argument

16.6.2.2 Execution of a CALL Statement.

Execution of a CALL statement results in

(1) evaluation of actual arguments that are expressions,

(2) association of actual arguments with the corresponding dummy arguments, and

(3) the actions specified by the referenced subroutine.

Return of control from the referenced subroutine completes execution of the CALL statement.

A subroutine specified in a subprogram may be referenced within any other procedure subprogram or the main program of the executable program. A subprogram must not reference itself, either directly or indirectly.

When a CALL statement is executed, the referenced subroutine must be one of the subroutines specified in subroutine subprograms or by other means in the executable program.

16.6.2.3 Actual Arguments for a Subroutine.

The actual arguments in a subroutine reference must agree in order and number with the corresponding dummy arguments in the dummy argument list of the referenced subroutine. They must also agree in type except that an actual argument of type Boolean may have a corresponding dummy argument of type integer or real, and an actual argument of type integer or real may have a corresponding dummy argument of type Boolean. The use of a subroutine name or an alternate return specifier as an actual argument is an exception to the rule requiring agreement of type.

An actual argument in a subroutine reference must be one of the following:

(1) An expression except a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses unless the operand is the symbolic name of a constant

(2) An array name

(3) An intrinsic function name

(4) An external procedure name

(5) A dummy procedure name

(6) An alternate return specifier, of the form *s, where s is the statement label of an executable statement that appears in the same program unit as the CALL statement (16.8.3)

(7) An array expression

(8) An array section name

Note that an actual argument in a subroutine reference may be a dummy argument name that appears in a dummy argument list within the subprogram containing the reference. An asterisk dummy argument must not be used as an actual argument in a subprogram reference.

16.6.3 SUBROUTINE SUBPROGRAM RESTRICTIONS.

A SUBROUTINE statement must appear only as the first statement of a subroutine subprogram. A subroutine subprogram may contain any other statement except a BLOCK DATA, FUNCTION, or PROGRAM statement.

The symbolic name of a subroutine is a global name (19.1.1) and must not be the same as any other global name, except a common block name, or any local name in the program unit.

In a subroutine subprogram, the symbolic name of a dummy argument is local to the program unit and must not appear in an EQUIVALENCE, PARAMETER, SAVE, INTRINSIC, DATA, or COMMON statement, except as a common block name.

A character dummy argument whose length specification is an asterisk in parentheses must not appear as an operand for concatenation, except in a character assignment statement (10.4).

16.7 ENTRY STATEMENT

An ENTRY statement permits a procedure reference to begin with a particular executable statement within the function or subroutine subprogram in which the ENTRY statement appears. It may appear anywhere within a function subprogram after the FUNCTION statement or within a subroutine subprogram after the SUBROUTINE statement, except that an ENTRY statement must not appear between a block IF statement and its corresponding END IF statement, between a block WHERE and its corresponding END WHERE, or between a DO statement and the terminal statement of its DO-loop.

Optionally, a subprogram may have one or more ENTRY statements.

An ENTRY statement is classified as a nonexecutable statement.

16.7.1 FORM OF AN ENTRY STATEMENT.

The form of an ENTRY statement is:

ENTRY *en* [([*d* [,*d*]...])]

where: *en* is the symbolic name of an entry in a function or subroutine subprogram and is called an *entry name*. If the ENTRY statement appears in a subroutine subprogram, *en* is a *subroutine name*. If the ENTRY statement appears in a function subprogram, *en* is an *external function name*.

d is a variable name, array name, or dummy procedure name, or is an asterisk. *d* is a dummy argument. An asterisk is permitted in an ENTRY statement only in a subroutine subprogram.

Note that if there are no dummy arguments, either of the forms *en* or *en*() may be used in the ENTRY statement. A function that is specified by either form must be referenced by the form *en*() (16.2.1). A subroutine that is specified by either form may be referenced by a CALL statement of the form CALL *en* or CALL *en*().

The entry name *en* in a function subprogram may appear in a type-statement.

16.7.2 REFERENCING AN EXTERNAL PROCEDURE BY AN ENTRY NAME

An entry name in an ENTRY statement in a function subprogram identifies an external function within the executable program and may be referenced as an external function (16.5.2). An entry name in an ENTRY statement in a subroutine subprogram identifies a subroutine within the executable program and may be referenced as a subroutine (16.6.2).

When an entry name *en* is used to reference a procedure, execution of the procedure begins with the first executable statement that follows the ENTRY statement whose entry name is *en*.

An entry name is available for reference in any program unit of an executable program, except in the program unit that contains the entry name in an ENTRY statement.

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 or SUBROUTINE statement and other ENTRY statements in the same subprogram. However, each reference to a function or subroutine must use an actual argument list that agrees in order, number, and type with the dummy argument list in the corresponding FUNCTION, SUBROUTINE, or

ENTRY statement. The use of a subroutine name or an alternate return specifier as an actual argument is an exception to the rule requiring agreement of type.

16.7.3 ENTRY ASSOCIATION.

Within a function subprogram, all variables whose names are also the names of entries are associated with each other and with the variable, if any, whose name is also the name of the function subprogram (18.1.3). Therefore, any such variable that becomes defined causes all associated variables of the same type to become defined and all associated variables of different type to become undefined. Such variables are not required to be of the same type unless the type is character, but the variable whose name is used to reference the function must be in a defined state when a RETURN or END statement is executed in the subprogram. An associated variable of a different type must not become defined during the execution of the function reference.

16.7.4 ENTRY STATEMENT RESTRICTIONS.

Within a subprogram, an entry name must not appear both as an entry name in an ENTRY statement and as a dummy argument in a FUNCTION, SUBROUTINE, or ENTRY statement and must not appear in an EXTERNAL statement.

In a function subprogram, a variable name that is the same as an entry name must not appear in any statement that precedes the appearance of the entry name in an ENTRY statement, except in a type-statement.

If an entry name in a function subprogram is of type character, each entry name and the name of the function subprogram must be of type character. If the name of the function subprogram or any entry in the subprogram has a length of (*) declared, all such entities must have a length of (*) declared; otherwise, all such entities must have a length specification of the same integer value.

In a subprogram, a name that appears as a dummy argument in an ENTRY statement must not appear in an executable statement preceding that ENTRY statement unless it also appears in a FUNCTION, SUBROUTINE, or ENTRY statement that precedes the executable statement.

In a subprogram, a name that appears as a dummy argument in an ENTRY statement must not appear in the expression of a statement function statement unless the name is also a dummy argument of the statement function, appears in a FUNCTION or SUBROUTINE statement, or appears in an ENTRY statement that precedes the statement function statement.

If a dummy argument appears in an executable statement, the

execution of the executable statement is permitted during the execution of a reference to the function or subroutine only if the dummy argument appears in the dummy argument list of the procedure name referenced. Note that the association of dummy arguments with actual arguments is not retained between references to a function or subroutine.

An ENTRY statement must not appear in an array-valued function subprogram.

16.8 RETURN STATEMENT

A RETURN statement in a function subprogram or subroutine subprogram causes return of control to the referencing program unit. Execution of a RETURN statement in a main program terminates the execution of the executable program.

16.8.1 FORM OF A RETURN STATEMENT.

The form of a RETURN statement in a function subprogram or a main program is:

```
          RETURN
```

The form of a RETURN statement in a subroutine subprogram is:

```
          RETURN [e]
```

where _e_ is a scalar arithmetic or Boolean expression. If _e_ is not of type integer, the value of INT(_e_) is used.

16.8.2 EXECUTION OF A RETURN STATEMENT.

Execution of a RETURN statement terminates the reference of a function or subroutine subprogram. Such subprograms may contain more than one RETURN statement; however, a subprogram need not contain a RETURN statement. Execution of an END statement in a function or subroutine subprogram has the same effect as executing a RETURN statement in the subprogram.

In the execution of an executable program, a function or subroutine subprogram must not be referenced a second time without the prior execution of a RETURN or END statement in that procedure.

Execution of a RETURN statement in a function subprogram causes return of control to the currently referencing program unit. The value of the function (16.5) must be defined and is available to the referencing program unit.

Execution of a RETURN statement in a subroutine subprogram causes return of control to the currently referencing program unit. Return

of control to the referencing program unit completes execution of the CALL statement.

Execution of a RETURN statement terminates the association between the dummy arguments of the external procedure in the subprogram and the current actual arguments.

16.8.3 ALTERNATE RETURN.

If *e* is not specified in a RETURN statement, or if the value of *e* is less than one or greater than the number of asterisks in the SUBROUTINE or subroutine ENTRY statement that specifies the currently referenced name, control returns to the CALL statement that initiated the subprogram reference and this completes the execution of the CALL statement.

If $1 \le e \le n$, where *n* is the number of asterisks in the SUBROUTINE or subroutine ENTRY statement that specifies the currently referenced name, the value of *e* identifies the *e*th asterisk in the dummy argument list. Control is returned to the statement identified by the alternate return specifier in the CALL statement that is associated with the *e*th asterisk in the dummy argument list of the currently referenced name. This completes the execution of the CALL statement.

16.8.4 DEFINITION STATUS.

Execution of a RETURN statement (or END statement) within a subprogram causes all entities within the subprogram to become undefined, except for the following:

(1) Entities specified by SAVE statements

(2) Entities in blank common

(3) Initially defined entities that have neither been redefined nor become undefined

(4) Entities in a named common block that appears in the subprogram and appears in at least one other program unit that is referencing, either directly or indirectly, the subprogram

Note that if a named common block appears in the main program, the entities in the named common block do not become undefined at the execution of any RETURN statement in the executable program.

16.9 ARGUMENTS AND COMMON BLOCKS

Arguments and common blocks provide means of communication between the referencing program unit and the referenced procedure.

Data may be communicated to a statement function or intrinsic function by an argument list. Data may be communicated to and from an external procedure by an argument list or common blocks. Procedure names may be communicated to an external procedure only by an argument list.

A dummy argument appears in the argument list of a procedure. An actual argument appears in the argument list of a procedure reference.

The number of actual arguments must be the same as the number of dummy arguments in the procedure referenced.

16.9.1 DUMMY ARGUMENTS.

Statement functions, function subprograms, and subroutine subprograms use dummy arguments to indicate the types of actual arguments and whether each argument is a single value, array of values, procedure, or statement label. Note that a statement function dummy argument may be only a variable.

Each dummy argument is classified as a variable, array, dummy procedure, or asterisk. Dummy argument names may appear wherever an actual name of the same class (Section 19) and type may appear, except where they are explicitly prohibited.

Dummy argument names must not appear in EQUIVALENCE, DATA, PARAMETER, SAVE, INTRINSIC, or COMMON statements, except as common block names. A dummy argument name must not be the same as the procedure name appearing in a FUNCTION, SUBROUTINE, ENTRY, or statement function statement in the same program unit.

A dummy argument must not appear as the host array in an IDENTIFY statement.

16.9.2 ACTUAL ARGUMENTS

Actual arguments specify the entities that are to be associated with the dummy arguments for a particular reference of a subroutine or function. An actual argument must not be the name of a statement function in the program unit containing the reference. Actual arguments may be constants, symbolic names of constants, function references, scalar or array expressions involving operators, vector-valued section selectors, and scalar or array expressions enclosed in parentheses if and only if the associated dummy argument is not defined during execution of the referenced external procedure.

The type of each actual argument must agree with the type of its associated dummy argument, except when the actual argument is a subroutine name (16.9.3.4) or an alternate return specifier

(16.6.2.3).

16.9.3 ASSOCIATION OF DUMMY AND ACTUAL ARGUMENTS.

At the execution of a function or subroutine reference, an association is established between the corresponding dummy 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 within a function or subroutine subprogram of a dummy argument whose name appears in the dummy argument list of the procedure name referenced become associated with the actual argument when a reference to the function or subroutine is executed.

A valid association occurs only if the type of the actual argument is the same as the type of the corresponding dummy argument. A subroutine name has no type and must be associated with a dummy procedure name. An alternate return specifier has no type and must be associated with an asterisk.

If an actual argument is an expression, it is evaluated just before the association of arguments takes place.

If an actual argument is an array element name, its subscript is evaluated just before the association of arguments takes place. Note that the subscript value remains constant as long as that association of arguments persists, even if the subscript contains variables that are redefined during the association.

If an actual argument is a character substring name, its substring expressions are evaluated just before the association of arguments takes place. Note that the value of each of the substring expressions remains constant as long as that association of arguments persists, even if the substring expression contains variables that are redefined during the association.

If an actual argument is an external procedure name, the procedure must be available at the time a reference to it is executed.

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 variable appearing in the adjustable array declarator is not currently associated with an actual argument and is not in a common block.

Argument association may be carried through more than one level of 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 the program unit. Note that there is no retention of argument association between one reference of a subprogram and the next reference of the subprogram.

If an elemental intrinsic function is referenced with an array name, array section name, or array expression as an actual argument, the function will be implicitly referenced n times where n is the size of the actual argument array name, array section name, or array expression. Each implicit reference will associate one of the array elements of the array, array section, or array expression actual argument with the corresponding variable dummy argument. Every array element of the actual argument will be associated during the n implicit references. The order in which the associations are made is not specified. If there is more than one array, array section, or array expression as an actual argument, each implicit reference will associate the same relative array element of each array, array section, or array expression. For example, in the reference to the elemental intrinsic function ATAN2

```
INTEGER A(10), B(-5:4)
B = ATAN2 ( A, B )
```

one of the implicit references to ATAN2 will associate A(1) and B(-5) with the corresponding dummy arguments, another implicit reference will associate A(6) and B(0) with the corresponding dummy arguments, etc.

16.9.3.1 Length of Character Dummy and Actual Arguments

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 length len of a dummy argument of type character is less than the length of an associated actual argument, the leftmost len characters of the actual argument are associated with the dummy argument.

If a dummy argument of type character is an array name, the restriction on length is for the entire array and not for each array element. The length of an array element in the dummy argument array may be different from the length of an array element in an associated actual argument array, array element, or array element substring, but the dummy argument array must not extend beyond the end of the associated actual argument array.

If an actual argument is a character substring, the length of the actual argument is the length of the substring. If an actual argument is the concatenation of two or more operands, its length is the sum of the lengths of the operands.

16.9.3.2 Variables as Dummy Arguments.

A dummy argument that is a variable may be associated with an actual argument that is a variable, array element, substring, or expression.

If the actual argument is a variable name, array element name, or substring name, the associated dummy argument may be defined or redefined within the subprogram. If the actual argument is a constant, a symbolic name of a constant, a function reference, an expression involving operators, or an expression enclosed in parentheses, the associated dummy argument must not be redefined within the subprogram.

16.9.3.3 Arrays as Dummy Arguments

Within a program unit, the array declarator given for an array provides all array declarator information needed for the array in an execution of the 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 only if the actual argument is an array name or array element name.

The number and size of dimensions in an actual argument array section or array expression involving operators or parentheses must agree with the number and size of the dimensions in an associated dummy argument array declarator.

Permitted associations between actual and dummy arguments are shown by the following table:

ACTUAL ARGUMENT / DUMMY ARGUMENT	ARRAY NAME	ARRAY SECTION/ EXPRESSION	ARRAY ELEMENT	ASSUMED SIZE ARRAY NAME	ASSUMED SHAPE ARRAY NAME
CONSTANT ARRAY	YES	YES Same Shape	YES	YES	YES Same Shape
ADJUSTABLE ARRAY	YES	YES Same Shape	YES	YES	YES Seme Shape
ASSUMED SIZE ARRAY	YES	NO	YES	YES	NO
ASSUMED SHAPE ARRAY	YES Same Shape	YES Same Shape	NO	NO	YES Same Shape
VARIABLE	NO	NO	YES	NO	NO

A "YES" entry indicates that the association is permitted. If "Same Shape" is also specified, then the actual and dummy argument must have the same shape.

If the actual argument is a noncharacter array name, the size of the dummy argument array must not exceed the size 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. Note that association by array elements exists for character arrays if there is egreement in length between the actual argument and the dummy argument array elements; if the lengths do not agree, the dummy and actual argument array elements do not consist of the same characters, but an association still exists.

If the actual argument is a noncharacter array element name, the size of the dummy argument array must not exceed the size of the actual argument array plus one minus the subscript value of the array element. When an actual argument is a noncharacter array element name with a subscript value of as, the dummy argument array element with a subscript value of ds becomes associated with the actual argument array element that has a subscript value of as + ds - 1 (Table 1, 5.5.3).

If the actual argument is a character array name, character array element name, or character array element substring name and begins at character storage unit acu of an array, character storage unit dcu of an associated dummy argument array becomes associated with character storage unit acu + dcu - 1 of the actual argument array.

16.9.3.4 Procedures as Dummy Arguments.

A dummy argument that is a dummy procedure may be associated only with an actual argument that is an intrinsic function, external function, subroutine, or another dummy procedure.

If a dummy argument is used as if it were an external function, the associated actual argument must be an intrinsic function, external function, or dummy procedure. A dummy argument that becomes associated with an intrinsic function never has any automatic typing property, even if the dummy argument name appears in Table 5 (16.10). 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. If a dummy argument name is used as if it were an external function and that name also appears in Table 5, the intrinsic function corresponding to the dummy argument name is not available for referencing within the subprogram.

A dummy argument that is used as a procedure name in a function reference and is associated with an intrinsic function must have arguments that agree in order, number, and type with those specified in Table 5 for the intrinsic function.

If a dummy argument appears in a type-statement and an EXTERNAL statement, the actual argument must be the name of an intrinsic function, external function, or dummy procedure.

If the dummy argument is referenced as a subroutine, the actual argument must be the name of a subroutine or dummy procedure and must not appear in a type-statement or be referenced as a function.

Note that it may not be possible to determine in a given program unit whether a dummy procedure is associated with a function or a subroutine. If a procedure name appears only in a dummy argument list, an EXTERNAL statement, and an actual argument list, it is not possible to determine whether the symbolic name becomes associated with a function or subroutine by examination of the subprogram alone.

16.9.3.5 Asterisks as Dummy Arguments.

A dummy argument that is an asterisk may appear only in the dummy argument list of a SUBROUTINE statement or an ENTRY statement in a subroutine subprogram.

A dummy argument that is an asterisk may be associated only with an actual argument that is an alternate return specifier in the CALL statement that identifies the current referencing name. If a dummy argument is an asterisk, the corresponding actual argument must be an alternate return specifier.

16.9.3.6 Restrictions_on_Association_of_Entities.

If a subprogram reference causes a dummy argument in the referenced subprogram to become associated with another dummy argument in the referenced subprogram, neither dummy argument may become defined during execution of that subprogram. For example, if a subroutine is headed by

```
SUBROUTINE XYZ (A,B)
```

and is referenced by

```
CALL XYZ (C,C)
```

then the dummy arguments A and B each become associated with the same actual argument C and therefore with each other. Neither A nor B may become defined during this execution of subroutine XYZ or by any procedures referenced by XYZ.

If a subprogram reference causes a dummy argument to become associated with an entity in a common block in the referenced subprogram or in a subprogram referenced by the referenced subprogram, neither the dummy argument nor the entity in the common block may become defined within the subprogram or within a subprogram referenced by the referenced subprogram. For example, if a subroutine contains the statements:

```
SUBROUTINE XYZ (A)
COMMON C
```

and is referenced by a program unit that contains the statements:

```
COMMON B
CALL XYZ (B)
```

then the dummy argument A becomes associated with the actual argument B, which is associated with C, which is in a common block. Neither A nor C may become defined during execution of the subroutine XYZ or by any procedures referenced by XYZ.

16.9.4 COMMON_BLOCKS.

A common block provides a means of communication between external procedures or between a main program and an external 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 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 the datum is in a defined state of the same type as the type of the name used to reference the datum. However, an integer variable that has been assigned a statement label must not be referenced in any program unit other than the one in which it was assigned (10.3).

No difference in data type is permitted between the defined state and the type of the reference, except that either part of a complex datum may be referenced also as a real datum.

In a subprogram that has declared a named common block, the entities in the 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 are specified by SAVE statements and those that were initially defined by DATA statements and have neither been redefined nor become undefined.

Execution of a RETURN or END statement does not cause entities in blank common or in any named common block that appears in the main program to become undefined.

Common blocks may be used also to reduce the total number of storage units required for an executable program by causing two or more subprograms to share some of the same storage units. This sharing of storage is permitted if the rules for defining and referencing data are not violated.

16.10 TABLE 5 INTRINSIC FUNCTIONS

Table 5
Elemental Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Type Conversion	Conversion to Integer int(a) See Note 1	1	INT	-	Integer	Integer
				INT	Real	Integer
				IFIX	Real	Integer
				IDINT	Double	Integer
				IHINT	Half	Integer
				-	Complex	Integer
	Conversion to Real See Note 2	1	REAL	REAL	Integer	Real
				FLOAT	Integer	Real
				-	Real	Real
				SNGL	Double	Real
				EXTEND	Half	Real
				-	Complex	Real
	Conversion to Double See Note 3	1	DBLE	-	Integer	Double
				-	Real	Double
				-	Double	Double
				-	Complex	Double
	Conversion to Complex See Note 4	1 or 2	CMPLX	-	Integer	Complex
				-	Real	Complex
				-	Double	Complex
				-	Complex	Complex
	Conversion to Integer See Note 5	1		ICHAR	Character	Integer
	Conversion to Character See Note 5	1		CHAR	Integer	Character
	Conversion to Half	1	HALF	-	Integer	Half
				-	Real	Half
				-	Double	Half
				-	Half	Half
				-	Complex	Half

Table 5 (continued)
Elemental Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
	Conversion to Boolean See note 14	1	BOOL	--	Integer	Boolean
				--	Real	Boolean
				--	Double	Boolean
				--	Half	Boolean
				--	Complex	Boolean
				--	Character	Boolean
	Conversion to Logical	1		BTOL	Bit	Logical
	Conversion to Bit	1		LTOB	Logical	Bit
Truncation	int(a) See Note 1	1	AINT	AINT	Real	Real
				DINT	Double	Double
				HINT	Half	Half
Nearest Whole Number	int(a+.5) if a≥0	1	ANINT	ANINT	Real	Real
	int(a-.5) if a<0	1		DNINT	Double	Double
				HNINT	Half	Half
Nearest Integer	int(a+.5) if a≥0	1	NINT	NINT	Real	Integer
	int(a-.5) if a<0	1		IDNINT	Double	Integer
				IHNINT	Half	Integer
Absolute Value	|a|	1	ABS	IABS	Integer	Integer
				ABS	Real	Real
	See Note 6			DABS	Double	Double
				HABS	Half	Half
	(ar**2+ai**2)**.5	1		CABS	Complex	Real

Table 5 (continued)
Elemental Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Remaindering	a1-int(a1/a2)*a2 See Note 1	2	MOD	MOD	Integer	Integer
				AMOD	Real	Real
				DMOD	Double	Double
				HMOD	Half	Half
Transfer of Sign	|a1| if a2 ≥ 0	2	SIGN	ISIGN	Integer	Integer
	-|a1| if a2 < 0			SIGN	Real	Real
				DSIGN	Double	Double
				HSIGN	Half	Half
Positive Difference	a1-a2 if a1>a2	2	DIM	IDIM	Integer	Integer
	0 if a1≤a2			DIM	Real	Real
				DDIM	Double	Double
				HDIM	Half	Half
Extended Precision Product	a1*a2	2		DPROD	Real	Double
				RPROD	Half	Real
Choosing Largest Value	max(a1,a2,...)	≥2	MAX	MAX0	Integer	Integer
				AMAX1	Real	Real
				DMAX1	Double	Double
				HMAX1	Half	Half
				AMAX0	Integer	Real
				MAX1	Real	Integer

16.0 FUNCTIONS AND SUBROUTINES
16.10 TABLE 5 INTRINSIC FUNCTIONS

Table 5 (continued)
Elemental Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Choosing Smallest Value	min(a1,a2,...)	≥2	MIN	MIN0 AMIN1 DMIN1 HMIN1	Integer Real Double Half	Integer Real Double Half
				AMIN0 MIN1	Integer Real	Real Integer
Length	Length of Character Entity	1		LEN	Character	Integer
Index of a Substring	Location of Substring a2 in String a1 See Note 10	2		INDEX	Character	Integer
Imaginary Part of Complex Argument	ai See Note 6	1		AIMAG	Complex	Real
Conjugate of a Complex Argument	(ar,-ai) See Note 6	1		CONJG	Complex	Complex
Square Root	(a)**1/2	1	SQRT	SQRT DSQRT HSQRT CSQRT	Real Double Half Complex	Real Double Half Complex

Table 5 (continued)
Elemental Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Exponential	e**a	1	EXP	EXP	Real	Real
				DEXP	Double	Double
				HEXP	Half	Half
				CEXP	Complex	Complex
Natural Logarithm	log(a)	1	LOG	ALOG	Real	Real
				DLOG	Double	Double
				HLOG	Half	Half
				CLOG	Complex	Complex
Common Logarithm	log10(a)	1	LOG10	ALOG10	Real	Real
				DLOG10	Double	Double
				HLOG10	Half	Half
Sine	sin(a)	1	SIN	SIN	Real	Real
				DSIN	Double	Double
				HSIN	Half	Half
				CSIN	Complex	Complex
Cosine	cos(a)	1	COS	COS	Real	Real
				DCOS	Double	Double
				HCOS	Half	Half
				CCOS	Complex	Complex
Tangent	tan(a)	1	TAN	TAN	Real	Real
				DTAN	Double	Double
				HTAN	Half	Half
Cotangent	cotan(a)	1	COTAN	COTAN	Real	Real
				HCOTAN	Half	Half

Table 5 (continued)
Elemental Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Arcsine	arcsin(a)	1	ASIN	ASIN	Real	Real
				DASIN	Double	Double
				HASIN	Half	Half
Arccosine	arccos(a)	1	ACOS	ACOS	Real	Real
				DACOS	Double	Double
				HACOS	Half	Half
Arctangent	arctan(a)	1	ATAN	ATAN	Real	Real
				DATAN	Double	Double
				HATAN	Half	Half
	arctan(a1/a2)	2	ATAN2	ATAN2	Real	Real
				DATAN2	Double	Double
				HATAN2	Half	Half
Hyperbolic Sine	sinh(a)	1	SINH	SINH	Real	Real
				DSINH	Double	Double
				HSINH	Half	Half
Hyperbolic Cosine	cosh(a)	1	COSH	COSH	Real	Real
				DCOSH	Double	Double
				HCOSH	Half	Half
Hyperbolic Tangent	tanh(a)	1	TANH	TANH	Real	Real
				DTANH	Double	Double
				HTANH	Half	Half

Table 5 (continued)
Elemental Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Lexically Greater Than or Equal	a1 ≥ a2 See Note 12	2		LGE	Character	Logical
Lexically Greater Than	a1 > a2 See Note 12	2		LGT	Character	Logical
Lexically Less Than or Equal	a1 ≤ a2 See Note 12	2		LLE	Character	Logical
Lexically Less Than	a1 < a2 See Note 12	2		LLT	Character	Logical
Random Number Generator	Random number in range (0.,1.) See Note 13.	0 or 1		RANF		Real
Shift	Boolean result of a1 shifted a2 bit positions See Note 15	2		SHIFT	a1: any but char,bit,or half a2: integer	Boolean
Mask	Boolean result of a left-justified 1 bits See Note 16	1		MASK	Integer	Boolean
Error Function	erf(a)	1		ERF	Real	Real
Complementary Error Function	erfc(a) See Note 17	1		ERFC	Real	Real
Hyperbolic Arctangent	atanh(a)	1		ATANH	Real	Real
Sine	sind(a), where a is in degrees	1		SIND	Real	Real
Cosine	cosd(a), where a is in degrees	1		COSD	Real	Real
Tangent	tand(a), where a is in degrees	1		TAND	Real	Real

Table 5 (continued)
Elemental Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Boolean Product	Boolean result of .AND. operator See Note 33	any>=2		AND	any but bit or char Bit	Boolean Bit
Boolean sum	Boolean result of .OR. operator See Note 33	any>=2		OR	any but bit or char Bit	Boolean Bit
Exclusive Or	Boolean result of .XOR. operator See Note 33	any>=2		XOR	any but bit or char Bit	Boolean Bit
Nonequivalence	Same as Exclusive Or See Note 33	any>=2		NEQV	any but bit or char Bit	Boolean Bit
Equivalence	Boolean result of .EQV. operator See Note 33	any>=2		EQV	any but bit or char Bit	Boolean Bit
Complement	Boolean result of .NOT. operator See Note 33	1		COMPL	any but bit or char Bit	Boolean Bit
Extract Bits	extb(a,i,j) See Note 18	3		EXTB	a: See Note 18 i: integer j: integer	Boolean
Insert Bits	insb(a,i,j,b) See Note 19	4		INSB	a and b: See Note 19 i: integer j: integer	Boolean
Merge under mask	See Note 20	3	MERGE		a1,a2 int a1,a2 Real a1,a2 Dbl a1,a2 Half a1,a2 Cplx a1,a2 Log a1,a2 Char a1,a2 Bit	Integer Real Double Half Complex Logical Character Bit

Table 5 (continued)
Array-valued Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Dot Product	sum(conjg(a1)*a2) See Note 21	2	DOTPRODUCT		Integer Real Double Half Complex	Integer Real Double Half Complex
Matrix Transpose	Row and column interchange of a See Note 22	1	TRANSPOSE		Integer Real Double Half Complex Logical Character Bit	Integer Real Double Half Complex Logical Character Bit
Summation	Sum of all the elements of a1 along dimension a2 See Note 32	1, 2, or 3	SUM		Integer Real Double Half Complex	Integer Real Double Half Complex
Product	Product of all the elements of a1 along dimension a2 See Note 32	1, 2, or 3	PRODUCT		Integer Real Double Half Complex	Integer Real Double Half Complex
Value of Maximum Element of an array	Maximum of all elements of a1 along dimension a2. See Note 32	1, 2, or 3	MAXVAL		Integer Real Double Half	Integer Real Double Half
Value of Minimum Element of an array	Minimum of all elements of a1 along dimension a2. See Note 32	1, 2, or 3	MINVAL		Integer Real Double Half	Integer Real Double Half
Count number of true values	Count of the number of true elements of a1 along dimension a2. See Note 23	1 or 2	COUNT		Logical	Integer

Table 5 (continued)
Array-valued Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Find any true value	.TRUE. if any element of a1 is true along dimension a2, .FALSE. if no elements of a1 are true along dimension a2 See Note 23	1 or 2	ANY		Logical	Logical
Find all true values	.TRUE. if all elements of a1 are true along dimension a2, .FALSE. if any element of a1 is false along dimension a2 See Note 23	1 or 2	ALL		Logical	Logical
Matrix Multiply	Matrix Multiply a1 by a2 See Note 24	2	MATMUL		Integer Real Double Half Complex Logical	Integer Real Double Half Complex Logical
Number of dimensions of an array	See Note 25	1	RANK		Integer Real Double Half Complex Logical Character Bit Boolean	Integer Integer Integer Integer Integer Integer Integer Integer Integer

Table 5 (continued)
Array-valued Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Pack Array into Vector	See Note 30	2 or 3	PACK		a1,a3 Int	Integer
					a1,a3 Real	Real
					a1,a3 Dble	Double
					a1,a3 Half	Half
					a1,a3 Cplx	Cplx
					a1,a3 Log	Logical
					a1,a3 Char	Character
					a1,a3 Bit	Bit
					a1,a3 Bool	Boolean
Unpack Vector Into Array	See Note 31	3	UNPACK		a1,a3 Int	Integer
					a1,a3 Real	Real
					a1,a3 Dble	Double
					a1,a3 Half	Half
					a1,a3 Cplx	Complex
					a1,a3 Log	Logical
					a1,a3 Char	Character
					a1,a3 Bit	Bit
					a1,a3 Bool	Boolean
Create Diagonal Array	See Note 34	2	DIAGONAL		a1,a3 Int	Integer
					a1,a3 Real	Real
					a1,a3 Dble	Double
					a1,a3 Half	Half
					a1,a3 Cplx	Complex
					a1,a3 Log	Logical
					a1,a3 Char	Character
					a1,a3 Bit	Bit
					a1,a3 Bool	Boolean
Sequence of Integers	See Note 35	2 or 3	SEQ		Integer	Integer
Logical Alternation	See Note 36	3	ALT		Integer	Logical

Table 5 (continued)
Array-valued Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Size of Array Dimensions	See Note 26	1 or 2	EXTENT		a1 Integer	Integer
					a1 Real	Integer
					a1 Double	Integer
					a1 Half	Integer
					a1 Complex	Integer
					a1 Logical	Integer
					a1 Char	Integer
					a1 Bit	Integer
					a1 Boolean	Integer
Size of Array	See Note 27	1	SIZE		Integer	Integer
					Real	Integer
					Double	Integer
					Half	Integer
					Complex	Integer
					Logical	Integer
					Character	Integer
					Bit	Integer
					Boolean	Integer
Lower Bound of an Array	See Note 28	1 or 2	LBOUND		a1 Integer	Integer
					a1 Real	Integer
					a1 Double	Integer
					a1 Half	Integer
					a1 Complex	Integer
					a1 Logical	Integer
					a1 Char	Integer
					a1 Bit	Integer
					a1 Boolean	Integer
Upper Bound of an Array	See Note 29	1 or 2	UBOUND		a1 Integer	Integer
					a1 Real	Integer
					a1 Double	Integer
					a1 Half	Integer
					a1 Complex	Integer
					a1 Logical	Integer
					a1 Char	Integer
					a1 Bit	Integer
					a1 Boolean	Integer

Table 5 (continued)
Array-valued Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Convert Rowwise Array into Columnwise Array	See Note 37	1	ROWWISE		Integer Real Double Half Complex Logical Character Bit Boolean	Integer Real Double Half Complex Logical Character Bit Boolean
Convert Columnwise Array into Rowwise Array	See Note 38	1	COLUMNWISE		Integer Real Double Half Complex Logical Character Bit Boolean	Integer Real Double Half Complex Logical Character Bit Boolean
Increase Rank of an array	See Note 39	3	SPREAD		a1 Integer a1 Real a1 Dble a1 Half a1 Complex a1 Logical a1 Char a1 Bit a1 Bool	Integer Integer Integer Integer Integer Integer Integer Integer Integer
Replicate an array dimension	See Note 40	3	REPLICATE		a1 Integer a1 Real a1 Dble a1 Half a1 Complex a1 Logical a1 Char a1 Bit a1 Bool	Integer Integer Integer Integer Integer Integer Integer Integer Integer

Notes for Table 5:

(1) For a of type integer, int(a)=a. For a of type real, half precision, or double precision, there are two cases: if $|a| < 1$, int(a)=0; if $|a| \geq 1$, int(a) is the integer whose magnitude is the largest integer that does not exceed the magnitude of a and whose sign is the same as the sign of a. For example,

int(-3.7) = -3

For a of type complex, int(a) is the value obtained by applying the above rule to the real part of a.

For a of type real, IFIX(a) is the same as INT(a).

For a of type Boolean, INT(a) is the integer represented by the bit string a.

For a of type half, INT(a)=INT(REAL(A)).

(2) For a of type real, REAL(a) is a. For a of type integer, double precision, or half precision, REAL(a) is as much precision of the significant part of a as a real datum can contain. For a of type complex, REAL(a) is the real part of a.

For a of type integer, FLOAT(a) is the same as REAL(a).

For a of type Boolean, REAL(a) is the real datum represented by the bit string a.

(3) For a of type double precision, DBLE(a) is a. For a of type integer or real, DBLE(a) is as much precision of the significant part of a as a double precision datum can contain. For a of type complex, DBLE(a) is as much precision of the significant part of the real part of a as a double precision datum can contain.

For a of type half or Boolean, DBLE(a)=DBLE(REAL(a)).

(4) CMPLX may have one or two arguments. If there is one argument, it may be of type integer, real, double precision, or complex. If there are two arguments, they must both be of the same type and may be of type integer, real, or double precision.

For a of type complex, CMPLX(a) is a. For a of type integer,

real, or double precision, CMPLX(a) is the complex value whose real part is REAL(a) and whose imaginary part is zero.

CMPLX(a1,a2) is the complex value whose real part is REAL(a1) and whose imaginary part is REAL(a2).

For a of type half or Boolean, CMPLX(a)=CMPLX(REAL(a)).

(5) ICHAR provides a means of converting from a character to an integer, based on collating weight of the character in the collating weight table used by the processor. The first character in the collating sequence corresponds to position 0 and the last to position n-1, where n is the number of characters in the collating sequence.

The value of ICHAR(a) is an integer in the range: 0 ≤ ICHAR(a) ≤ n-1, where a is an argument of type character of length one. The value of a must be a character capable of representation in the processor. The collating weight of that character is the value of ICHAR.

For any characters c1 and c2 capable of representation in the processor, (c1 .LE. c2) is true if and only if (ICHAR(c1) .LE. ICHAR(c2)) is true, and (c1 .EQ. c2) is true if and only if (ICHAR(c1) .EQ. ICHAR(c2)) is true.

CHAR(i) returns a character having collating weight i in the collating weight table used by the processor (3.1.5). If more than one character has collating weight i, the choice of character returned is made by the processor. The value is of type character of length one. i must be an integer expression whose value must be in the range 0 ≤ i ≤ n-1.

ICHAR(CHAR(i)) = i for 0 ≤ i ≤ n-1.

CHAR(ICHAR(c)) = c for any character c capable of representation in the processor.

(6) 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.

(7) All angles are expressed in radians.

(8) The result of a function of type complex is the principal value.

(9) All arguments in an intrinsic function reference must be of the same type.

(10) INDEX(a1,a2) returns an integer value indicating the starting

position within the character string a1 of a substring identical to string a2. If a2 occurs more than once in a1, the starting postion of the first occurence is returned.

If a2 does not occur in a1, the value zero is returned. Note that zero is returned if LEN(a1) < LEN(a2).

(11) The value of the argument of the LEN function need not be defined at the time the function reference is executed.

(12) LGE(a1,a2) returns the value true if a1=a2 or if a1 follows a2 in the collating sequence described in American National Standard Code for Information Interchange, ANSI X3.4-1977 (ASCII), and otherwise returns the value false.

LGT(a1,a2) returns the value true if a1 follows a2 in the collating sequence described in ANSI X3.4-1977 (ASCII), and otherwise returns the value false.

LLE(a1,a2) returns the value true if a1=a2 or if a1 precedes a2 in the collating sequence described in ANSI X3.4-1977 (ASCII), and otherwise returns the value false.

LLT(a1,a2) returns the value true if a1 precedes a2 in the collating sequence described in ANSI X3.4-1977 (ASCII), and otherwise returns the value false.

If the operands for LGE, LGT, LLE, and LLT 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.

If either of the character entities being compared contains a character that is not in the ASCII character set, the result is processor-dependent.

(13) The values are uniformly distributed in the range (0.,1.).

If one argument is present, it determines the shape of the result returned. For example, RANF(B) where B has shape (4,3) will return an array of random numbers with shape (4,3) and RANF(X) where X is a scalar will return a scalar.

(14) For a of type integer, real, or Boolean, BOOL(a) is the bit string constituting the datum. For a of type double precision, half precision, or complex, BOOL(a) = BOOL (REAL(a)). For a of type character, BOOL(a) is the Boolean datum having the value of the Hollerith constant

nHa

where:

$\underline{n}$ is the integer having the value of MIN($\underline{g}$, LEN($\underline{a}$)), where $\underline{g}$ is the maximum number of characters representable in internal processor code in a Boolean datum

$\underline{f}$ is the string of the leftmost $\underline{n}$ characters of $\underline{a}$.

(15) The shift is left circular if $\underline{a}2 \geq 0$, and right with sign extension and end off if $\underline{a}2<0$. For $\underline{a}1$ of type integer, real, logical or Boolean, the bit string constituting the datum is shifted. For $\underline{a}1$ of type double precision or complex, the bit string constituting REAL ($\underline{a}1$) is shifted.

(16) The string of $\underline{a}$ 1 bits occupies the leftmost $\underline{a}$ bit positions if $0<\underline{a}\leq\underline{g}$, where $\underline{g}$ is the number of bit positions in one numeric storage unit. The fill is a string of 0 bits. If $\underline{a}=0$, all bits in the result are 0 bits.

(17) erfc($\underline{a}$) = 1-erf($\underline{a}$)

(18) EXTB extracts bits from its first argument, as indicated by its second and third arguments. The first argument may be of any data type except character or bit. For a half precision, double precision, or complex first argument, the argument is converted to REAL($\underline{a}$).

Arguments $\underline{i}$ and $\underline{j}$ must be of type integer. Argument $\underline{i}$ indicates the first bit to be extracted, numbering from bit zero on the left. Argument $\underline{j}$ indicates the number of bits to be extracted.

(19) INSB inserts bits from its first argument, as indicated by its second and third arguments, into a copy of its fourth argument. The fourth argument itself is not altered. The first and fourth arguments may be of any data type except character or bit. For a half precision, double precision, or complex first or fourth argument, the argument is converted to REAL($\underline{a}$) or REAL($\underline{b}$) respectively.

Arguments $\underline{i}$ and $\underline{j}$ must be of type integer. The result is the value of argument $\underline{b}$ except that $\underline{j}$ bits beginning with bit $\underline{i}$ are replaced with the rightmost $\underline{j}$ bits from argument $\underline{a}$. Bits are numbered from the left beginning with zero.

(20) MERGE produces a result which contains the value from $\underline{a}1$ when the corresponding value in $\underline{a}3$ is true, otherwise it contains the value from $\underline{a}2$. $\underline{a}3$ must be a scalar or array expression of type logical. $\underline{a}1$ and $\underline{a}2$ must be of the same type and they must be conformable with $\underline{a}3$. If $\underline{a}3$ is a scalar, $\underline{a}1$ and $\underline{a}2$ must be scalars. $\underline{a}1$ and $\underline{a}2$ may be scalars or array

expressions. The result of MERGE is of the same type as a1 and a2 and has the same shape as a3.

(21) a1 and a2 must be one dimensional array expressions and must have the same shape.

(22) a must be a two dimensional array expression. If the shape of a is (m,n), the shape of the result of TRANSPOSE will be (n,m) with the row and column values interchanged.

(23) If a2 is omitted, the function is applied to all elements of a1 to yield a scalar value. a1 must be an array, array section, or an array expression. If a1 has size zero then the result has size zero, unless dimension a2 of a1 is the only one of size zero in which case the result has non-zero size and all elements have the same value.

(24) The operation performed by MATMUL(a1,a2) depends on the shapes of a1 and a2. If a1 and a2 are of arithmetic type, the three possible operations are:

(1) a1 and a2 are two dimensional array expressions; the shape of a1 is (n,m) and the shape of a2 is (m,k). The shape of the result array is (n,k). The value of the (i,h) element of the result array is:

SUM (a1(i, *) * a2(*, h))

(2) a1 is a one dimensional array expression and a2 is a two dimensional array expression; the shape of a1 is (m), the shape of a2 is (m,k). The shape of the result array is (k). The value of the ith element of the result array is:

SUM (a1(*) * a2(*, i))

(3) a1 is a two dimensional array expression and a2 is a one dimensional array expression; the shape of a1 is (n,m), the shape of a2 is (m). The shape of the result array is (n). The value of the ith element of the result array is:

SUM (a1(i, *) * a2(*))

The formulas for the values of the three possible operations performed by a matrix multiply mathematically define an interpretation for the MATMUL function. The formulas do not specify the computational method the processor must use to compute MATMUL.

If a1 and a2 are of type logical, the three possible

operations are:

(1) a1 and a2 are two dimensional array expressions; the shape of a1 is (n,m) and the shape of a2 is (m,k). The shape of the result array is (n,k). The value of the (i,h) element of the array is:

ANY (a1(i,*) .AND. a2(*, h))

(2) a1 is a one dimensional array expression and a2 is a two dimensional array expression; the shape of a1 is (m), the shape of a2 is (m,k). The shape of the result array is (k). The value of the ith element of the result array is:

ANY (a1(*) .AND. a2(*,i))

(3) a1 is a two dimensional array expression and a2 is a one dimensional array expression; the shape of a1 is (n,m), the shape of a2 is (m). The shape of the result array is (n). The value of the ith element of the result array is:

ANY (a1(i, *) .AND. a2(*))

(25) RANK(a) returns the number of dimensions of a as an integer value. a may be a scalar, array, array section, or array expression. If a is a scalar, then RANK(a) will return zero. RANK(a) where a is an array section returns the number of section selectors in the array section reference a. RANK(a) where a is an allocatable array returns the number of dimensions of a, whether or not a is allocated.

(26) If a1 is an array, array section, or array expression with n dimensions, EXTENT(a1) returns a one dimensional array of n integer values. The value of the ith element of the result of EXTENT(a1) is the size of the ith dimension of a1. The result of EXTENT(a1) is undefined if a1 is an assumed size array.

EXTENT(a1,a2) returns the size of the dimension of a1 corresponding to the value of a2. The result is an integer scalar value. a2 must be an integer scalar value such that $1 \leq a2 \leq n$. The result of EXTENT(a1,a2) is undefined if a1 is an assumed size array and a2 designates the last dimension. The result of EXTENT(a1,a2) is undefined if a1 is an allocatable array that is not currently allocated.

(27) SIZE(a1) returns the size of the array, array section, or array expression a1 an an integer. The size is the product of the sizes of each of the dimensions. The result of SIZE

is undefined if a1 is an assumed size array or if a1 is an allocatable array that is not currently allocated.

(28) If a1 is an array, array section, or array expression with n dimensions, then LBOUND(a1) returns a one dimensional array of n integer values. The value of the ith element of the result of LBOUND(a1) is the lower bound of the ith dimension of a1. LBOUND(a1,a2) returns as an integer scalar value the lower dimension bound of the dimension of a1 corresponding to the value of a2 where 1 ≤ a2 ≤ n. The lower bound of an array section or an array expression is defined to be 1. The result of LBOUND is undefined if a1 is an allocatable array that is not currently allocated.

(29) If a1 is an array, array section, or array expression with n dimension, then UBOUND(a1) returns a one dimensional array of n integer values. The value of the ith element of the result of UBOUND(a1) is the upper bound of the ith dimension of a1. UBOUND(a1,a2) returns as an integer scalar value the upper bound of the dimension of a1 corresponding to the value of a2 where 1 ≤ a2 ≤ n. The result of UBOUND(a1) is undefined if a1 is an assumed size array. The result of UBOUND(a1,a2) is undefined if a1 is an assumed size array and a2 designates the last dimension. The upper bound of an array section or an array expression is defined to be the number of elements in the dimension. The result of UBOUND(a1) or UBOUND(a1,a2) is undefined if a1 is an allocatable array that is not currently allocated.

(30) Pack Array Into Vector. PACK returns a one-dimensional array of the same type as a1 consisting of all elements of a1 corresponding to true elements of a2. The elements of a1 are taken in subscript order. a2 must be a logical array expression conformable with a1. If a3 is not present, the result array's size is the number of true elements in a2. If a3 is present, the result array size is the same as a3. a3 is a one-dimensional array of the same type as a1 with at least as many elements as there are true elements in a2.

(31) Unpack Vector into Array. UNPACK returns an array of the same type as a1 and the same shape as a2. a2 is a logical array. a3 is an array of the same type as a1 and the same shape as a2. Elements of the result array corresponding to true values of a2 are taken in subscript order and assigned successive values of the one-dimensional array a1. Each element of the result array corresponding to a false value of a2 is assigned the value of the corresponding element of a3. a2 and a3 must be conformable arrays. The number of elements in the vector a1 must be greater than or equal to the number of true values in array a2. a3 may be a scalar. In this case, each element of the result array corresponding to a

false value of a2 is assigned the value of a3.

(32) a1 must be an array, array section or array expression. The type of the result will be the same as the type of a1.

The reduction dimension may be specified by the optional keyword DIM =.

The mask may be specified by the optional keyword MASK =.

If a2 (DIM) is omitted and a mask (a3) is specified, the MASK = keyword is required.

If a3 is not specified, it is as if it were specified as an array conforming to a1 and having all elements true.

If a2 is not specified, the function is applied to all of the elements of a1 corresponding to true elements of a3 to yield a scalar; if a1 has size zero or if a3 has no true elements then the value is as given in the table below.

If a2 is specified, the function is applied along dimension a2 to elements of a1 corresponding to true elements of a3 to yield an array with rank RANK(a1)-1; whenever the masking by a3 leaves no elements, the corresponding component of the result is as given in the table below; if a1 has size zero then the result has size zero unless dimension a2 of a1 is the only one of size zero, in which case the result has non-zero size and all elements have the value given in the table below.

Function	Result when the number of elements is zero
SUM	zero
PRODUCT	one
MAXVAL	the smallest valid machine number of type a1
MINVAL	the largest valid machine number of type a1

(33) For any argument ai of type integer, real, logical, or Boolean the bit string that is operated on is the datum itself.

For any argument ai of type double precision, half precision, or complex the bit string that is operated on is REAL(ai).

If the arguments are type bit, then the function result is also type bit. If any argument is type bit, then all arguments must be type bit.

(34) DIAGONAL(a1,a2) returns a two dimensional array, call it diag, with shape (a2,a2) of the same type as a1. a1 is a scalar or a one dimensional array expression with a2 array elements. a2 is an integer scalar value. If a1 is a one dimensional array object, its size must be equal to a2. If diag(s,s) for 1≤s≤a2 is considered as a one dimensional array of size a2, its value is a1. All other elements of diag have a default value according to the type of a1. Default values by type are:

TYPE	VALUE
Integer	0
Real	0.0
Double precision	0.0D0
Half precision	0.0
Complex	(0.,0)
Logical	.FALSE.
Character	m blanks where LEN(a1) is m
Bit	B"0"

(35) SEQ produces a one dimensional integer array with size s given by MAX((a2-a1+a3)/a3 , 1). The array values are a1, a1+a3, a1+2*a3,...,a1+(s-1)*a3. a1 and a2 must be integer scalar values. a3, if present, must be a nonzero integer scalar value. If a3 is omitted, it defaults to one.

(36) ALT produces a one dimensional logical array of size a3 consisting of (a3/(a1+a2))+1 repetitions of a1 false values followed by a2 true values truncated on the right to size a3. a1, a2, and a3 must be positive integer scalar values.

(37) ROWWISE returns as its result the rowwise array corresponding to its actual argument columnwise array, such that each element (s1,...,sn) of the resulting rowwise array will have a value identical to element (s1,...,sn) of the actual argument.

(38) COLUMNWISE returns as its result the columnwise array corresponding to its actual argument rowwise array, such that each element (s1,...,sn) of the resulting columnwise array will have a value identical to element (s1,...,sn) of the actual argument.

(39) SPREAD increases the dimensionality of a1 by duplicating its value along the dimension specified by a2 a3 times. a1 may be a scalar (defined to have zero dimensions) or an array expression with n dimensions. a2 must be an integer value such that 1 ≤ a2 ≤ RANK(a1)+1. a3 must be an integer value such that a3>0.

The result type of SPREAD is the same as the type of a1. If a1 is a scalar, a2 must equal 1. The result is a one dimensional array of size a3 all of whose elements have the value a1. If a1 is an array with shape (d1,d2,...,dn) then the result of SPREAD(a1,i,a3) is an array, call it sp, with the shape of (d1,d2,...,dn) where the total number of dimensions is n+1 and di is the added dimension. The value of array section sp(*,*,...,si,...,*) where 1≤si≤a is given by a1.

(40) REPLICATE produces an array with the same number of dimensions as a1 and the same type as a1. a1 must be an array with dimension n where 1≤n≤7. a2 must be an integer value such that 1≤a2≤n. a3 must be an integer value such that a3>0. If a1 is an array expression with shape (d1,d2,...,di,...,dn) then the result of REPLICATE(a1,i,a3) is an array with shape (d1,d2,...,a3*di,...dn). The value of the result array is a3 copies of a1 concatenated to form the new ith dimension.

16.10.1 RESTRICTIONS ON RANGE OF ARGUMENTS AND RESULTS.

Restrictions on the range of arguments and results for intrinsic functions when referenced by their specific names are as follows:

(1) Remaindering: The result for MOD, AMOD, HMOD, and DMOD is undefined when the value of the second argument is zero.

(2) Transfer of Sign: If the value of the first argument of ISIGN, SIGN, HSIGN, or DSIGN is zero, the result is zero, which is neither positive nor negative (4.1.3).

(3) Square Root: The value of the argument of SQRT, HSQRT, and DSQRT must be greater than or equal to zero. The result of CSQRT is the principal value with the real part greater than or equal to zero. When the real part of the result is zero, the imaginary part is greater than or equal to zero.

(4) Logarithms: The value of the argument of ALOG, DLOG, HLOG, ALOG10, HLOG10, and DLOG10 must be greater than zero. The value of the argument of CLOG must not be (0.,0.). The range of the imaginary part of the result of CLOG is: -pi < imaginary part ≤ pi. The imaginary part of the result is pi only when the real part of the argument is less than zero and the imaginary part of the argument is zero.

(5) Sine, Cosine, and Tangent: The absolute value of the argument of SIN, DSIN, HSIN, COS, DCOS, HCOS, TAN, HTAN, and DTAN is not restricted to be less than 2 pi.

(6) Arcsine: The absolute value of the argument of ASIN, HASIN, and DASIN must be less than or equal to one. The range of

the result is: -pi/2 ≤ result ≤ pi/2.

(7) Arccosine: The absolute value of the argument of ACOS, HACOS, and DACOS must be less than or equal to one. The range of the result is: 0 ≤ result ≤ pi.

(8) Arctangent: The range of the result for ATAN, HATAN, and DATAN is: -pi/2 ≤ result ≤ pi/2. If the value of the first argument of ATAN2, HATAN2, or DATAN2 is positive, the result is positive. If the value of the first argument is zero, the result is zero if the second argument is positive and pi if the second argument is negative. If the value of the first argument is negative, the result is negative. If the value of the second argument is zero, the absolute value of the result is pi/2. The arguments must not both have the value zero. The range of the result for ATAN2, HATAN2, and DATAN2 is: -pi < result ≤ pi.

(9) Shift: The result for SHIFT is undefined when the absolute value of the second argument is greater than the number of bit positions in one numeric storage unit.

(10) Mask: The result for MASK is undefined when the argument is negative or is greater than the number of bit positions in one numeric storage unit.

(11) Hyberbolic Arctangent: The absolute value of the argument of ATANH must be less than one.

The above restrictions on arguments and results also apply to the intrinsic functions when referenced by their generic names.

16.11 PROCESSOR-SUPPLIED FUNCTIONS

The following functions are processor supplied but are not intrinsic functions. As such, they may be freely replaced by user functions or subroutines of the same name.

16.11.1 DATE

The form of a reference to DATE is:

```
DATE()
```

The DATE function returns the current date as a character string in the form 'yyyy-mm-dd' where yyyy represents the four digits of the year, mm represents the number of the month (01,02,...,12), and dd the number of the day of the month. DATE must be declared to be of type character with length 10 in the calling program.

16.11.2 TIME

The form of a reference to TIME is:

    TIME()

The time function returns the current reading of the system clock in the form 'hh:mm:ss', where hh is the hours from 00 to 23, mm is minutes from 00 to 59, and ss is seconds from 00 to 59. TIME must be declared character of length 8 in the calling program.

16.11.3 SECOND

The form of a reference to SECOND is:

    SECOND()

The SECOND function returns the number of CPU seconds used since the beginning of execution of the program as a value of type real.

16.11.4 IOCLAS

The form of a reference to IOCLAS is:

    IOCLAS(ios)

where: ios is an error code returned by IOSTAT=.

The IOCLAS function takes an integer value ios that is an input/output status code obtained by IOSTAT= on an I/O statement and classifies the code into a limited number of error classes. The integer value returned by the function is the error class number. The error classes and associated numbers are as follows:

n	Class of error
0	not an error (ios less than or equal to zero)
1	physical device error
2	invalid formatted, list-directed, or NAMELIST input data
3	invalid format specification
4	incorrect value in a cilist, olist, cllist, or iulist of an I/O statement
5	operation prohibited on file
6	exceeded record size
7	attempt to read past end of file
8	write operation followed by read on a sequential file
9	iolist calls for more data than in unformatted record
10	record manager detected but unclassifiable error.

11 output record limit exceeded on file

16.11.5 NUMERR

The form of reference to NUMERR is:

NUMERR()

NUMERR returns, as an integer value, the number of input errors since the last LIMERR call. See the description of LIMERR for details.

16.11.6 UNIT

The form of a reference to UNIT is:

UNIT(u,a,b)

where:

u is the unit specifier

a is the first variable or array element of the block of memory specified in the preceding BUFFER IN or OUT statement

b is the last variable or array element of the block of memory specified in the preceding BUFFER IN or OUT statement

Example:

```
BUFFER OUT(5,1) (B(1), B(100))
IF ( UNIT(5, B(1), B(100)) 12,14,16
```

Control transfers to the statement labeled 12, 14, or 16 if the value returned was -1., 0., or +1., repectively.

Parameters a and b of the UNIT function should be included so that the global optimizer can associate the call to the UNIT function with possible changes to the values of the entities between a and b. If UNIT is called with only the argument u (as in most older programs) the optimizer will be unable to detect that values between a and b are being modified asynchronously during the span of code between the BUFFER IN or BUFFER OUT and the corresponding UNIT call. Therefore, specification of all three arguments is recommended.

16.12 PROCESSOR-SUPPLIED SUBROUTINES

The following subroutines are processor-supplied but are not intrinsic routines. Therefore, they can be replaced by a user

subroutine or function of the same name without use of an EXTERNAL statement.

16.12.1 REMARK

The form of a call to REMARK is:

CALL REMARK(c)

where: c is a character expression.

The REMARK call places the character string c in the job log as a message. Messages exceeding a maximum system dependent length will be truncated.

16.12.2 CONNEC

The form of a call to CONNEC is:

CALL CONNEC(u,cs)

where: u is an integer expression specifying an external unit number that is connected to a file

cs is an integer expression specifying a character set.

A call to CONNEC causes the specified unit u to be associated with the terminal device if the job is executing interactively. If the job is not executing interactively, the unit u will be associated with a device in the same manner as if the call to CONNEC had not appeared. When a call to CONNEC is made, the contents of the file connected to the unit u become undefined. Input/output operations on a unit associated with an interactive terminal are restricted to formatted I/O, including list directed and NAMELIST. A unit that is connected to a direct access file may not be specified in a CONNEC call.

A cs value of one specifies input/output in the 128 character ASCII set defined by the X3.4 standard. A cs value of two specifies input/output in a transparent mode where the character set employed is terminal dependent but generally reflects the native character set of the device. Results are undefined for any other value of cs.

16.12.3 DISCON

The form of a call to DISCON is:

CALL DISCON(u)

where: u is an integer expression specifying an external unit number

A call to DISCON disassociates a unit and file from a terminal device. The call to DISCON has no effect if the unit u is not associated with a terminal. After execution of the reference to DISCON, the unit u is connected to a file of the same name as before but the file is now on a system selected device, normally mass storage. The contents of the file are undefined.

16.12.4 LIMERR

The form of a call to LIMERR is:

```
    CALL LIMERR(lim)
```

where: lim is an integer expression specifying the maximum number of formatted input errors allowed before program execution is terminated abnormally.

The LIMERR call allows formatted data input without risk of abnormal termination when improper input data is encountered. When LIMERR is used, program execution is not terminated until the number of errors occurring after the call exceeds the value of lim.

Use of LIMERR will inhibit termination of execution for input data errors detected during formatted input including list directed and NAMELIST input. LIMERR has no effect on the processing of errors in data input from a connected (terminal) file.

After a call to LIMERR, the error limit continues in effect for all subsequent READ statements until lim errors have been detected. LIMERR may be called more than once to reinitialize the error count and reset the value of lim. The current error count may be obtained by a reference to the function NUMERR. A LIMERR call with a value of zero for lim will cause the error limit to be set to zero and normal processor error handling behavior to be restored.

When improper input data is detected in a formatted or NAMELIST read with fewer than lim errors detected, the bad data field is bypassed, the count of errors detected is incremented by one, and processing continues with the next field. When improper input data is detected in a list-directed read, the count of errors detected is incremented by one, and control passes to the next executable statement following the list-directed READ statement. The value of the entity corresponding to the field in error is undefined.

16.12.5 RANSET

The form of a call to RANSET is:

    CALL RANSET(seed)

where seed is a value of type real, Boolean, or integer used to specify the starting seed for the random number function RANF.

16.12.6 RANGET

The form of a call to RANGET is:

    CALL RANGET(sval)

where sval is the value of the current seed of the random number generating function RANF. sval must be a variable or array element of type real, Boolean, or integer. The value of sval may be passed to RANSET at a later time to regenerate the same sequence of random numbers with RANF. Results are undefined if the value of sval is used in any other manner.

17.0 BLOCK DATA SUBPROGRAM

Block data subprograms are used to provide initial values for variables and array elements in named common blocks.

A block data subprogram is a program unit that has a BLOCK DATA statement as its first statement. A block data subprogram is nonexecutable. There may be more than one block data subprogram in an executable program.

17.1 BLOCK DATA STATEMENT

The form of a BLOCK DATA statement is:

BLOCK DATA [sub]

where sub is the symbolic name of the block data subprogram in which the BLOCK DATA statement appears.

The optional name sub is a global name (19.1.1) and must not be the same as the name of an external procedure, main program, or other block data subprogram in the same executable program. The name sub must not be the same as any local name in the subprogram.

17.2 BLOCK DATA SUBPROGRAM RESTRICTIONS

The BLOCK DATA statement must appear only as the first statement of a block data subprogram. The only other statements that may appear in a block data subprogram are IMPLICIT, PARAMETER, DIMENSION, ROWWISE, COMMON, SAVE, EQUIVALENCE, DATA, END, and type-statements. Note that comment lines are permitted.

If an entity in a named common block is initially defined, all entities having storage units in the common block storage sequence must be specified even if they are not all initially defined. More than one named common block may have entities initially defined in a single block data subprogram.

Only an entity in a named common block may be initially defined in a block data subprogram. Note that entities associated with an entity in a common block are considered to be in that common block.

The same named common block may not be specified in more than one block data subprogram in the same executable program.

There must not be more than one unnamed block data subprogram in an executable program. The name BLKDAT# is assigned to the unnamed block data subprogram.

18.0 ASSOCIATION AND DEFINITION

18.1 STORAGE AND ASSOCIATION

Storage sequences are used to describe relationships that exist among variables, array elements, substrings, common blocks, and arguments.

18.1.1 STORAGE SEQUENCE.

A *storage sequence* is a sequence (2.1) of storage units. The *size of a storage sequence* is the number of storage units in the storage sequence. A *storage unit* is a character storage unit, bit storage unit, or a numeric storage unit.

A variable or array element of type integer, real, Boolean, or logical has a storage sequence of one numeric storage unit.

A variable or array element of type double precision or complex has a storage sequence of two numeric storage units. In a complex storage sequence, the real part has the first storage unit and the imaginary part has the second storage unit.

A variable, array element, or substring of type character has a storage sequence of character storage units. The number of character storage units in the storage sequence is the length of the character entity. The order of the sequence corresponds to the ordering of character positions (4.8).

A variable or array element of type half precision has a storage sequence of half of one numeric storage unit.

A variable or array element of type bit has a storage sequence of one bit storage unit.

Each array and common block has a storage sequence (5.3.5 and 8.3.2).

18.1.2 ASSOCIATION OF STORAGE SEQUENCES

Two storage sequences s1 and s2 are *associated* if the ith storage unit of s1 is the same as the jth storage unit of s2. This causes the (i+k)th storage unit of s1 to be the same as the (j+k)th storage unit of s2, for each integer k such that $1 \le i+k \le$ size of s1 and $1 \le j+k \le$ size of s2.

18.1.3 ASSOCIATION OF ENTITIES.

Two variables, array elements, or substrings are _associated_ if their storage sequences are associated. Two entities are _totally associated_ if they have the same storage sequence. Two entities are _partially associated_ if they are associated but not totally associated.

The definition status and value of an entity affects the definition status and value of any associated entity. An EQUIVALENCE statement, a COMMON statement, an ENTRY statement (16.7.3), or a procedure reference (argument association) may cause association of storage sequences.

An EQUIVALENCE statement causes association of entities only within one program unit, unless one of the equivalenced entities is also in a common block (8.3).

Arguments and COMMON statements cause entities in one program unit to become associated with entities in another program unit (8.3 and 16.9). Note that association between actual and dummy arguments does not imply association of storage sequences except when the actual argument is the name of a variable, array element, array section, array, or substring.

In a function subprogram, an ENTRY statement causes the entry name to become associated with the name of the function subprogram which appears in the FUNCTION statement.

Partial association may exist between entities of different types.

Except for character entities, partial association may occur only through the use of COMMON, EQUIVALENCE, or ENTRY statements. Partial association must not occur through argument association, except for arguments of type character.

In the example:

```
REAL A(4),B
COMPLEX C(2)
DOUBLE PRECISION D
EQUIVALENCE (C(2),A(2),B), (A,D)
```

the third storage unit of C, the second storage unit of A, the storage unit of B, and the second storage unit of D are specified as the same. The storage sequences may be illustrated as:

```
   storage unit     ¦  1  ¦  2  ¦  3  ¦  4  ¦  5  ¦
                    ¦----C(1)---¦----C(2)---¦
                          ¦ A(1)¦ A(2)¦ A(3)¦ A(4)¦
                                ¦--B--¦
                          ¦-----D-----¦
```

A(2) and B are totally associated. The following are partially associated: A(1) and C(1), A(2) and C(2), A(3) and C(2), B and C(2), A(1) and D, A(2) and D, B and D, C(1) and D, and C(2) and D. Note that although C(1) and C(2) are each associated with D, C(1) and C(2) are not associated with each other.

Partial association of character entities occurs when some, but not all, of the storage units of the entities are the same. In the example:

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

A, B, and C are partially associated.

18.2 EVENTS THAT CAUSE ENTITIES TO BECOME DEFINED

Variables, arrays, array sections, array elements, and substrings become defined as follows:

(1) Execution of an arithmetic, logical, Boolean, bit, or character assignment statement causes each entity that precedes the equals to become defined.

(2) As execution of an input statement proceeds, each entity that is assigned a value of its corresponding type from the input medium becomes defined at the time of such assignment.

(3) Execution of a DO statement causes the DO-variable to become defined.

(4) Beginning of execution of action specified by an implied-DO list in an input/output statement causes the implied-DO-variable to become defined.

(5) A DATA statement causes entities to become initially defined at the beginning of execution of an executable program.

(6) Execution of an ASSIGN statement causes the variable in the statement to become defined with a statement label value.

(7) When an entity of a given type becomes defined, all totally associated entities of the same type become defined except that entities totally associated with the variable in an ASSIGN statement become undefined when the ASSIGN statement

is executed.

(8) A reference to a subprogram causes a dummy argument to become defined if the corresponding actual argument is defined with a value that is not a statement label value. Note that there must be agreement between the actual argument and the dummy argument (16.9.3).

(9) Execution of an input/output statement containing an input/output status specifier causes the specified integer variable or array element to become defined.

(10) Execution of an INQUIRE statement causes any entity that is assigned a value during the execution of the statement to become defined if no error condition exists.

(11) When a complex entity becomes defined, all partially associated real or Boolean entities become defined.

(12) When both parts of a complex entity become defined as a result of partially associated real or complex entities becoming defined, the complex entity becomes defined.

(13) When all characters of a character entity become defined, the character entity becomes defined.

(14) When all elements of an array become defined, the array becomes defined.

(15) When all elements of an array section become defined, the array section becomes defined.

(16) When a Boolean entity becomes defined, all associated integer or real entities become defined. When an integer or real entity becomes defined, all associated Boolean entities become defined.

(17) When a double precision entity becomes defined, all partially associated Boolean entities become defined.

(18) If both parts of a double precision entity become defined as a result of partially associated Boolean entities becoming defined, the double precision entity becomes defined.

(19) Zero-sized arrays, array sections and substrings are always defined.

18.3 EVENTS THAT CAUSE ENTITIES TO BECOME UNDEFINED

Variables, arrays, array sections, array elements, and substrings become undefined as follows:

(1) All entities are undefined at the beginning of execution of an executable program except zero-sized arrays or those entities initially defined by DATA statements.

(2) When an entity of a given type becomes defined, all totally associated entities of different type become undefined. However, when an entity of type Boolean is associated with an entity of type integer or real, the Boolean entity does not become undefined when the integer or real entity is defined and the integer or real entity does not become undefined when the Boolean entity is defined.

(3) Execution of an ASSIGN statement causes the variable in the statement to become undefined as an integer. Entities that are associated with the variable become undefined.

(4) When an entity of type other than character becomes defined, all partially associated entities become undefined, with the following exceptions:

   (a) when an entity of type real or Boolean is partially associated with an entity of type complex, the complex entity does not become undefined when the real or Boolean entity becomes defined and the real or Boolean entity does not become undefined when the complex entity becomes defined

   (b) when an entity of type double precision and an entity of type Boolean are partially associated, the double precision entity does not become undefined when the Boolean entity is defined and the Boolean entity does not become undefined when the double precision entity becomes defined

   (c) when an entity of type complex is partially associated with another entity of type complex, definition of one entity does not cause the other to become undefined.

(5) When the evaluation of a function causes an argument of the function or an entity in common to become defined and if a reference to the function appears in an expression in which the value of the function is not needed to determine the value of the expression, then the argument or the entity in common becomes undefined when the expression is evaluated (6.6.1).

(6) The execution of a RETURN statement or an END statement within a subprogram causes all entities within the subprogram to become undefined except for the following:

(a) Entities in blank common

(b) Initially defined entities that have neither been redefined nor become undefined

(c) Entities specified by SAVE statements

(d) Entities in a named common block that appears in the subprogram and appears in at least one other program unit that is, either directly or indirectly, referencing the subprogram

(7) When an error condition or end-of-file condition occurs during execution of an input statement, all of the entities specified by the input list of the statement become undefined.

(8) Execution of a direct access input statement that specifies a record that has not been previously written causes all of the entities specified by the input list of the statement to become undefined.

(9) Execution of an INQUIRE statement may cause entities to become undefined (13.10.3).

(10) When any character of a character entity becomes undefined, the character entity becomes undefined.

(11) When an entity becomes undefined as a result of conditions described in (5) through (10), all totally associated entities become undefined and all partially associated entities of type other than character become undefined.

(12) When any array element becomes undefined, the array and any array sections containing that array become undefined. This does not imply that the undefinition of one array element causes any other array element to become undefined.

19.0 SCOPE AND CLASSES OF SYMBOLIC NAMES

A symbolic name consists of from one to thirty-one alphanumeric characters, the first of which must be a letter. Some sequences of characters, such as format edit descriptors and keywords that uniquely identify certain statements, for example, GO TO, READ, FORMAT, etc., are not symbolic names in such occurrences nor do they form the first characters of symbolic names in such occurrences.

19.1 SCOPE OF SYMBOLIC NAMES

The scope of a symbolic name is an executable program, a program unit, a PROGRAM statement, a statement function statement, or an implied-DO list in a DATA statement.

The name of the main program and the names of block data subprograms, external functions, subroutines, and common blocks have a scope of an executable program.

The names of variables, arrays, constants, statement functions, intrinsic functions, NAMELIST group names, and dummy procedures have a scope of a program unit.

The names of variables that appear as dummy arguments in a statement function statement have a scope of that statement.

The names of variables that appear as the DO-variable of an implied-DO in a DATA statement have a scope of the implied-DO list.

A symbolic unit name (15.1.1) or alternate unit name (15.1.2) has a scope of the PROGRAM statement in which it appears.

19.1.1 GLOBAL ENTITIES.

The main program, common blocks, subprograms, and external procedures are global entities of an executable program. A symbolic name that identifies a global entity must not be used to identify any other global entity in the same executable program, except that a common block name may be a program name or a subprogram name.

19.1.1.1 Classes of Global Entities.

A symbolic name in one of the following classes is a global entity in an executable program:

(1) Common block

(2) External function

(3) Subroutine

(4) Main program

(5) Block data subprogram

19.1.2 LOCAL ENTITIES.

The symbolic name of a local entity identifies that entity in a single program unit. Within a program unit, a symbolic name that is in one class of entities local to the program unit must not also be in another class of entities local to the program unit. However, a symbolic name that identifies a local entity may, in a different program unit, identify an entity of any class that is either local to that program unit or global to the executable program. A symbolic name that identifies a global entity in a program unit must not be used to identify a local entity in that program unit, except for a common block name, an external function name (19.2.2), and a symbolic unit name or alternate unit name (19.2.13).

19.1.2.1 Classes of Local Entities.

A symbolic name in one of the following classes is a local entity in a program unit.

(1) Array

(2) Variable

(3) Constant

(4) Statement function

(5) Intrinsic function

(6) Dummy procedure

(7) NAMELIST group name

A symbolic name that is a dummy argument of a procedure is classified as a variable, array, or dummy procedure. The specification and usage must not violate the respective class rules.

19.2 CLASSES OF SYMBOLIC NAMES

In a program unit, a symbolic name must not be in more than one class except as noted in the following paragraphs of this section. There are no restrictions on the appearances of the same symbolic name in different program units of an executable program other than those noted in this section.

19.2.1 COMMON BLOCK

A symbolic name is the name of a common block if and only if it appears as a block name in a COMMON statement (8.3).

A common block name is global to the executable program.

A common block name in a program unit may also be the name of any local entity other than a constant, intrinsic function, or a local variable that is also an external function in a function subprogram. If a name is used for both a common block and a local entity, the appearance of that name in any context other than as a common block name in a COMMON or SAVE statement identifies only the local entity. Note that an intrinsic function name may be a common block name in a program unit that does not reference the intrinsic function.

19.2.2 EXTERNAL FUNCTION

A symbolic name is the name of an external function if it meets any of the following conditions:

(1) The name appears immediately following the word FUNCTION in a FUNCTION statement or the word ENTRY in an ENTRY statement within a function subprogram.

(2) It is not an array name, character variable name, statement function name, intrinsic function name, dummy argument, or subroutine name and every appearance is immediately followed by a left parenthesis except in a type-statement, in an EXTERNAL statement, or as an actual argument.

(3) The name appears immediately follwing the word FUNCTION in a FUNCTION statement in a procedure interface information block.

In a function subprogram, the name of a function that appears immediately after the word FUNCTION in a FUNCTION statement or immediately after the word ENTRY in an ENTRY statement may also be the name of a variable in that subprogram (16.5.1). At least one such function name must be the name of a variable in a function subprogram.

An external function name is global to the executable program.

19.2.3 SUBROUTINE

A symbolic name is the name of a subroutine if it meets either of the following conditions:

(1) The name appears immediately following the word SUBROUTINE in a SUBROUTINE statement or the word ENTRY in an ENTRY

statement within a subroutine subprogram.

(2) The name appears immediately following the word CALL in a CALL statement and is not a dummy argument.

A subroutine name is global to the executable program.

19.2.4 MAIN_PROGRAM

A symbolic name is the name of a main program if and only if it appears in a PROGRAM statement in the main program.

A main program name is global to the executable program.

19.2.5 BLOCK_DATA_SUBPROGRAM

A symbolic name is the name of a block data subprogram if and only if it appears in a BLOCK DATA statement.

A block data subprogram name is global to the executable program.

19.2.6 ARRAY

A symbolic name is the name of an array if it appears as the array name in an array declarator (5.2) in a DIMENSION, ROWWISE, COMMON, or type-statement.

An array name is local to a program unit.

An array name may be the same as a common block name.

19.2.7 VARIABLE

A symbolic name is the name of a variable if it meets all of the following conditions:

(1) It does not appear in a PARAMETER, INTRINSIC, or EXTERNAL statement.

(2) It is not the name of an array, subroutine, main program, or block data subprogram.

(3) It appears other than as the name of a common block, the name of an external function in a FUNCTION statement, or an entry name in an ENTRY statement in an external function.

(4) It is never immediately followed by a left parenthesis unless it is immediately preceded by the word FUNCTION in a FUNCTION statement, is immediately preceded by the word ENTRY in an ENTRY statement, or is at the beginning of a character substring name (5.10.1).

A variable name in the dummy argument list of a statement function statement is local to the statement function statement in which it occurs. Note that the use of a name that appears in Table 5 as a dummy argument of a statement function removes it from the class of intrinsic functions. A variable name that appears as an implied-DO-variable in a DATA statement is local to the implied-DO list. All other variable names are local to a program unit.

A statement function dummy argument name may also be the name of a variable or common block in the program unit. The appearance of the name in any context other than as a dummy argument of the statement function identifies the local variable or common block. The statement function dummy argument name and local variable name have the same type and, if of type character, both have the same constant length.

The name of an implied-DO-variable in a DATA statement may also be the name of a variable or common block in the program unit. The appearance of the name in any context other than as an implied-DO-variable in the DATA statement identifies the local variable or common block. The implied- DO-variable and the local variable have the same type.

19.2.8 CONSTANT

A symbolic name is the name of a constant if it appears as a symbolic name in a PARAMETER statement.

The symbolic name of a constant is local to a program unit.

19.2.9 STATEMENT FUNCTION

A symbolic name is the name of a statement function if a statement function statement (16.4) is present for that symbolic name and it is not an array name.

A statement function name is local to a program unit. A statement function name may be the same as a common block name.

19.2.10 INTRINSIC FUNCTION

A symbolic name is the name of an intrinsic function if it meets all of the following conditions:

(1) The name appears in the Specific Name column or the Generic Name column of Table 5.

(2) It is not an array name, statement function name, subroutine name, or dummy argument name.

(3) Every appearance of the symbolic name, except in an INTRINSIC

statement, a type-statement, or as an actual argument, is immediately followed by an actual argument list enclosed in parentheses.

An intrinsic function name is local to a program unit.

19.2.11 DUMMY_PROCEDURE

A symbolic name is the name of a dummy procedure if the name appears in the dummy argument list of a FUNCTION, SUBROUTINE, or ENTRY statement and meets one or more of the following conditions:

(1) It appears in an EXTERNAL statement.

(2) It appears immediately following the word CALL in a CALL statement.

(3) It is not an array name or character variable name, and every appearance is immediately followed by a left parenthesis except in a type-statement, in an EXTERNAL statement, in a CALL statement, as a dummy argument, as an actual argument, or as a common block name in a COMMON or SAVE statement.

A dummy procedure name is local to a program unit.

19.2.12 NAMELIST_GROUP_NAME

A symbolic name is a NAMELIST group name if and only if it appears as a group name in a NAMELIST statement.

A NAMELIST group name is local to a program unit.

A NAMELIST group name must not appear in any statement except a NAMELIST, READ, WRITE, PRINT, or PUNCH statement.

APPENDIX A - CHARACTER SET AND COLLATION

The table below lists the ASCII character set and shows the processor-defined weight tables available. The processor-defined user-specified weight tables are selected by parameter values supplied to utility subroutine COLSEQ (3.1.5). The defined weight tables are:

'ASCII'
'ASCII6'
'COBOL6'
'DISPLAY'
'STANDARD'.

Fixed and user-specified tables are predefined with 'ASCII' and 'DISPLAY' respectively. The 'INSTALL' parameter value selects 'COBOL6' values.

Hex Character Code	ASCII Graphic	ASCII	ASCII6	COBOL6	DISPLAY
		Weights			
00	NUL	0	0	0	45
01	SOH	1	0	0	45
02	STX	2	0	0	45
03	ETX	3	0	0	45
04	EOT	4	0	0	45
05	ENQ	5	0	0	45
06	ACK	6	0	0	45
07	BEL	7	0	0	45
08	BS	8	0	0	45
09	HT	9	0	0	45
0A	LF	10	0	0	45
0B	VT	11	0	0	45
0C	FF	12	0	0	45
0D	CR	13	0	0	45
0E	SO	14	0	0	45
0F	SI	15	0	0	45
10	DLE	16	0	0	45
11	DC1	17	0	0	45
12	DC2	18	0	0	45
13	DC3	19	0	0	45
14	DC4	20	0	0	45
15	NAK	21	0	0	45
16	SYN	22	0	0	45
17	ETB	23	0	0	45
18	CAN	24	0	0	45
19	EM	25	0	0	45
1A	SUB	26	0	0	45
1B	ESC	27	0	0	45
1C	FS	28	0	0	45
1D	GS	29	0	0	45
1E	RS	30	0	0	45
1F	US	31	0	0	45
20	SP	32	0	0	45
21	!	33	1	34	54
22	"	34	2	23	52

23	#	35	3	5	48
24	$	36	4	16	43
25	%	37	5	2	51
26	&	38	6	6	55
27	'	39	7	7	56
28	(	40	8	21	41
29	)	41	9	13	42
2A	*	42	10	17	39
2B	+	43	11	15	37
2C	,	44	12	20	46
2D	-	45	13	18	38
2E	.	46	14	12	47
2F	/	47	15	19	40
30	0	48	16	54	27
31	1	49	17	55	28
32	2	50	18	56	29
33	3	51	19	57	30
34	4	52	20	58	31
35	5	53	21	59	32
36	6	54	22	60	33
37	7	55	23	61	34
38	8	56	24	62	35
39	9	57	25	63	36
3A	:	58	26	53	0
3B	;	59	27	14	63
3C	<	60	28	24	58
3D	=	61	29	22	44
3E	>	62	30	9	59
3F	?	63	31	8	57
40	@	64	32	1	60
41	A	65	33	25	1
42	B	66	34	26	2
43	C	67	35	27	3
44	D	68	36	28	4
45	E	69	37	29	5
46	F	70	38	30	6
47	G	71	39	31	7
48	H	72	40	32	8
49	I	73	41	33	9
4A	J	74	42	35	10
4B	K	75	43	36	11
4C	L	76	44	37	12
4D	M	77	45	38	13
4E	N	78	46	39	14
4F	O	79	47	40	15
50	P	80	48	41	16
51	Q	81	49	42	17
52	R	82	50	43	18
53	S	83	51	45	19
54	T	84	52	46	20
55	U	85	53	47	21
56	V	86	54	48	22
57	W	87	55	49	23
58	X	88	56	50	24
59	Y	89	57	51	25

APPENDIX A - CHARACTER SET AND COLLATION

5A	Z	90	58	52	26
5B	[	91	59	3	49
5C	\	92	60	10	61
5D	]	93	61	44	50
5E	^	94	62	11	62
5F	_	95	63	4	53
60	`	96	32	1	60
61	a	97	33	25	1
62	b	98	34	26	2
63	c	99	35	27	3
64	d	100	36	28	4
65	e	101	37	29	5
66	f	102	38	30	6
67	g	103	39	31	7
68	h	104	40	32	8
69	i	105	41	33	9
6A	j	106	42	35	10
6B	k	107	43	36	11
6C	l	108	44	37	12
6D	m	109	45	38	13
6E	n	110	46	39	14
6F	o	111	47	40	15
70	p	112	48	41	16
71	q	113	49	42	17
72	r	114	50	43	18
73	s	115	51	45	19
74	t	116	52	46	20
75	u	117	53	47	21
76	v	118	54	48	22
77	w	119	55	49	23
78	x	120	56	50	24
79	y	121	57	51	25
7A	z	122	58	52	26
7B	{	123	59	3	49
7C	¦	124	60	10	61
7D	}	125	61	44	50
7E	~	126	62	11	62
7F	DEL	127	63	4	53

APPENDIX B - SUMMARY OF LANGUAGE EXTENSIONS

Below is a list of Control Data Common FORTRAN language features that constitute extensions to ANSI FORTRAN, 77.

Basic Concepts

1. Symbolic names can be up to 31 characters in length and may include an underscore.

2. Bit data type and machine word oriented bit manipulation (for types bit and Boolean)

Characters, Lines, and Execution Sequence

1. The characters " (quote) and underscore are added to character set

2. Collation control (for character relationals)

3. C$-directives

Data Types and Constants

1. Boolean type

2. Bit type

3. Half Precision type

4. Hollerith, octal, and hexadecimal constants

5. Symbolic constants as real and imaginary parts of a complex constant.

Arrays and Substrings

1. Boolean entities can appear in dimension bound expressions.

2. Real, half precision, double precision, complex, and Boolean expressions as subscript or substring expressions.

3. Array sections

4. Assumed shape dummy arrays

5. Allocatable arrays

6. Extend adjustable dimension bounds expressions to allow all entities except external function references and array element references

7. ALLOCATE and FREE statements

8. Rowwise arrays

Expressions

1. Boolean expressions

2. Boolean entities in arithmetic expressions

3. Double precision and complex operands can be combined using the +, -, *, and / operators.

4. A double precision operand can be raised to a complex power.

5. Boolean entities in relational expressions

6. .XOR. operator

7. Bit expressions

8. Bit entities in relational expressions

Specification Statements

1. BOOLEAN type statement

2. BIT type statement

3. HALF PRECISION type statement

4. INTERFACE and END INTERFACE statements

5. ROWWISE statement

6. Entities in named common blocks can be initialized (by means of a DATA statement) in any program unit.

7. Boolean, bit, and half precision types in IMPLICIT statement

8. Boolean, bit, and half precision types in PARAMETER statement

9. A variable may be specified as integer following its appearance in a dimension bound expression.

10. Numeric, character, and bit data may be mixed in common blocks and via EQUIVALENCE.

DATA Statement

1. Replicated value list

2. Boolean and Half precision entities

Assignment Statement

1. Boolean and Bit assignment statements

2. Multiple assignment statement

3. Array assignment statements

4. Extended logical assignment (i.e. logical=bit expressions)

Control Statements

1. Real, half precision, double precision, complex or Boolean expression in computed GO TO statement

2. Boolean expression in arithmetic IF statement

3. Boolean expression as DO-parameter

4. One-trip DO-loops are optional.

5. Extended-range DO-loops

6. Logical WHERE statement

7. Block WHERE, OTHERWISE, and ENDWHERE statements.

Input-Output Statements

1. NAMELIST statement

2. BUFFER IN and BUFFER OUT statements

3. ENCODE and DECODE statements

4. PUNCH statement

5. Record length specifier in OPEN statement for file accessed sequentially (means page width)

6. More than one unit can be connected to a single external file

7. Extended internal files

8. Boolean external unit identifier

9. Buffer length specifier in OPEN statement

10. A comma may optionally follow the output list of an output statement that specifies list-directed formatting.

11. Implicit connection.

Format Specification

1. Edit descriptors added

   a. "h(1)h(2)..."

   b. Rw

   c. Ow

   d. Ow.m

   e. Zw

   f. Zw.m

   g. Bw

2. Aw edit descriptor can be used for non-character data.

3. NAMELIST formatting

4. Format in noncharacter array

Functions and Subroutines

1. Statement functions

   a. Boolean, half precision, and bit statement functions

   b. Statement function reference may occur prior to its statement function statement.

   c. Conversion of actual argument to type of dummy argument for statement function reference.

   d. The expression in a statement function statement may have a substring reference as a primary.

2. External procedures

   a. Boolean arguments can be associated with integer or real arguments.

   b. An external procedure name can be the same as a common block name.

   c. A RETURN statement can appear in a main program.

   d. The expression e in RETURN e can be any arithmetic or Boolean expression, as contrasted with integer type only in the standard.

   e. Array valued functions

3. Intrinsic functions

   a. Type conversion

      i. INT, REAL, HALF, DBLE, and CMPLX can have Boolean arguments.

      ii. BOOL function

      iii. BTOL, LTOB, HALF added

   b. Boolean operations (AND, OR, XOR, NEQV, EQV, COMPL)

   c. Miscellaneous functions (SHIFT, MASK, RANF)

   d. Additional mathematical functions (ERF, ERFC, ATANH, SIND, COSD, TAND)

   e. Additional functions for half precision (HINT, HNINT, IHNINT, IHINT, HABS, ...)

   f. Array manipulation functions (MERGE, DOTPRODUCT, TRANSPOSE, ...)

Block_Data_Subprogram

1. The name of a block data subprogram may be the same as the name of a common block.

Association_and_Definition

1. Partial association may exist between a Boolean entity and a double precision or complex entity.

2. Association may exist between a Boolean entity and an integer or real entity.

3. Association may exist between character entities and noncharacter entities.

Table of Contents

1.0 INTRODUCTION 1-1
1.1 PURPOSE 1-1
1.2 PROCESSOR 1-1
1.3 SCOPE 1-1
1.3.1 INCLUSIONS. 1-1
1.3.2 EXCLUSIONS. 1-1
1.4 CONFORMANCE 1-2
1.5 NOTATION USED IN THIS STANDARD 1-3

2.0 FORTRAN TERMS AND CONCEPTS 2-1
2.1 SEQUENCE 2-1
2.2 SYNTACTIC ITEMS 2-1
2.3 STATEMENTS, COMMENTS, AND LINES 2-2
2.3.1 CLASSES OF STATEMENTS. 2-2
2.4 PROGRAM UNITS AND PROCEDURES 2-2
2.4.1 PROCEDURES. 2-2
2.4.2 EXECUTABLE PROGRAM. 2-3
2.5 VARIABLE 2-3
2.6 ARRAY 2-3
2.6.1 ARRAY ELEMENTS 2-3
2.6.2 ARRAY SECTIONS 2-4
2.7 SUBSTRING 2-4
2.8 SCALAR 2-4
2.9 DUMMY ARGUMENT 2-4
2.10 SCOPE OF SYMBOLIC NAMES AND STATEMENT LABELS 2-5
2.11 LIST 2-5
2.12 DEFINITION STATUS 2-5
2.13 REFERENCE 2-6
2.14 STORAGE 2-6
2.15 ASSOCIATION 2-7
2.16 SHAPE 2-7
2.17 CONFORMABILITY 2-8
2.18 ALLOCATABLE 2-8

3.0 CHARACTERS, LINES, AND EXECUTION SEQUENCE 3-1
3.1 FORTRAN CHARACTER SET 3-1
3.1.1 LETTERS 3-1
3.1.2 DIGITS. 3-1
3.1.3 ALPHANUMERIC CHARACTERS 3-2
3.1.4 SPECIAL CHARACTERS. 3-2
3.1.5 COLLATING SEQUENCE AND GRAPHICS. 3-2
3.1.6 BLANK CHARACTER. 3-3
3.2 LINES 3-3
3.2.1 COMMENT LINE 3-3
3.2.2 INITIAL LINE. 3-4
3.2.3 CONTINUATION LINE. 3-4
3.3 STATEMENTS 3-4
3.4 STATEMENT LABELS 3-4
3.5 ORDER OF STATEMENTS AND LINES 3-5
3.6 NORMAL EXECUTION SEQUENCE AND TRANSFER OF CONTROL 3-6
3.7 C$-DIRECTIVES 3-7
3.7.1 LISTING CONTROL 3-8

TABLE OF CONTENTS

3.7.1.1 List Option Switches 3-9
3.7.2 CONDITIONAL COMPILATION 3-9
3.7.2.1 Processor Control 3-11
3.7.3 COLLATION CONTROL 3-11
3.7.4 DO-LOOP CONTROL 3-12

4.0 DATA TYPES AND CONSTANTS 4-1
4.1 DATA TYPES 4-1
4.1.1 DATA TYPE OF A NAME. 4-1
4.1.2 TYPE RULES FOR DATA AND PROCEDURE IDENTIFIERS. 4-1
4.1.3 DATA TYPE PROPERTIES. 4-2
4.2 CONSTANTS 4-2
4.2.1 DATA TYPE OF A CONSTANT. 4-2
4.2.2 BLANKS IN CONSTANTS. 4-3
4.2.3 ARITHMETIC CONSTANTS. 4-3
4.2.3.1 Signs of Constants. 4-3
4.3 INTEGER TYPE 4-3
4.3.1 INTEGER CONSTANT. 4-3
4.4 REAL TYPE 4-3
4.4.1 BASIC REAL CONSTANT. 4-3
4.4.2 REAL EXPONENT. 4-4
4.4.3 REAL CONSTANT. 4-4
4.5 DOUBLE PRECISION TYPE 4-4
4.5.1 DOUBLE PRECISION EXPONENT. 4-4
4.5.2 DOUBLE PRECISION CONSTANT. 4-4
4.6 COMPLEX TYPE 4-5
4.6.1 COMPLEX CONSTANT 4-5
4.7 LOGICAL TYPE 4-5
4.7.1 LOGICAL CONSTANT. 4-5
4.8 CHARACTER TYPE 4-5
4.8.1 CHARACTER CONSTANT. 4-6
4.9 BOOLEAN TYPE 4-6
4.9.1 BOOLEAN CONSTANT 4-6
4.9.1.1 Hollerith Constant 4-6
4.9.1.2 Octal Constant 4-7
4.9.1.3 Hexadecimal Constant 4-8
4.10 HALF PRECISION TYPE 4-8
4.10.1 HALF PRECISION EXPONENT 4-8
4.10.2 HALF PRECISION CONSTANT 4-8
4.11 BIT TYPE 4-9
4.11.1 BIT CONSTANT 4-9

5.0 ARRAYS AND SUBSTRINGS 5-1
5.1 ARRAYS AND ARRAY SECTIONS 5-1
5.1.1 ARRAY NAME AND ARRAY SECTION REFERENCE 5-1
5.2 ARRAY DECLARATOR 5-1
5.2.1 FORM OF AN ARRAY DECLARATOR 5-2
5.2.1.1 Form of a Dimension Declarator 5-2
5.2.1.2 Value of Dimension Bounds 5-3
5.2.2 KINDS AND OCCURRENCES OF ARRAY DECLARATORS 5-3
5.2.2.1 Actual Array Declarator 5-3
5.2.2.2 Dummy Array Declarator 5-4
5.2.2.3 Allocatable Array Declarator 5-4
5.3 PROPERTIES OF AN ARRAY 5-4
5.3.1 DATA TYPE OF AN ARRAY AND AN ARRAY ELEMENT 5-4

TABLE OF CONTENTS

5.3.2 DIMENSIONS OF AN ARRAY 5-4
5.3.3 SIZE OF AN ARRAY 5-5
5.3.4 ARRAY ELEMENT ORDERING 5-5
5.3.5 ARRAY STORAGE SEQUENCE 5-6
5.4 ARRAY ELEMENT NAME 5-6
5.5 SUBSCRIPT 5-6
5.5.1 FORM OF A SUBSCRIPT 5-6
5.5.2 SUBSCRIPT EXPRESSION 5-7
5.5.3 SUBSCRIPT VALUE 5-7
5.6 ARRAY SECTION NAMES 5-10
5.6.1 SECTION SUBSCRIPT 5-10
5.6.1.1 Form of a Section Subscript 5-10
5.6.1.2 Section Subscript Expression 5-11
5.7 ARRAY SECTION 5-11
5.8 DUMMY AND ACTUAL ARRAYS 5-12
5.8.1 ADJUSTABLE ARRAYS AND ADJUSTABLE DIMENSIONS. 5-13
5.9 USE OF ARRAY NAMES 5-14
5.9.1 APPEARANCE OF ARRAY SECTION NAMES 5-14
5.10 CHARACTER SUBSTRING 5-15
5.10.1 SUBSTRING NAME. 5-15
5.10.2 SUBSTRING EXPRESSION. 5-16

6.0 EXPRESSIONS 6-1
6.1 ARITHMETIC EXPRESSIONS 6-1
6.1.1 ARITHMETIC OPERATORS. 6-1
6.1.2 FORM AND INTERPRETATION OF ARITHMETIC EXPRESSIONS . . 6-2
6.1.2.1 Primaries. 6-3
6.1.2.2 Factor. 6-3
6.1.2.3 Term. 6-4
6.1.2.4 Arithmetic Expression. 6-4
6.1.3 ARITHMETIC CONSTANT EXPRESSION. 6-5
6.1.3.1 Integer Constant Expression. 6-6
6.1.4 TYPE AND INTERPRETATION OF ARITHMETIC EXPRESSIONS . . 6-6
6.1.4.1 Boolean Operands and Arithmetic Operators 6-11
6.1.5 INTEGER DIVISION. 6-11
6.2 CHARACTER EXPRESSIONS 6-12
6.2.1 CHARACTER OPERATOR. 6-12
6.2.2 FORM AND INTERPRETATION OF CHARACTER EXPRESSIONS. . 6-12
6.2.2.1 Character Primaries. 6-13
6.2.2.2 Character Expression. 6-13
6.2.3 CHARACTER CONSTANT EXPRESSION. 6-14
6.3 RELATIONAL EXPRESSIONS 6-14
6.3.1 RELATIONAL OPERATORS. 6-14
6.3.2 ARITHMETIC RELATIONAL EXPRESSION. 6-15
6.3.3 INTERPRETATION OF ARITHMETIC RELATIONAL EXPRESSIONS . 6-15
6.3.4 CHARACTER RELATIONAL EXPRESSION. 6-15
6.3.5 INTERPRETATION OF CHARACTER RELATIONAL EXPRESSIONS . 6-15
6.3.6 BIT RELATIONAL EXPRESSION 6-16
6.3.7 INTERPRETATION OF BIT RELATIONAL EXPRESSIONS 6-16
6.4 LOGICAL EXPRESSIONS 6-16
6.4.1 LOGICAL OPERATORS. 6-17
6.4.2 FORM AND INTERPRETATION OF LOGICAL EXPRESSIONS. . . 6-17
6.4.2.1 Logical Primaries. 6-18
6.4.2.2 Logical Factor. 6-18
6.4.2.3 Logical Term. 6-18

TABLE OF CONTENTS

6.4.2.4 Logical Disjunct. 6-19
6.4.2.5 Logical Expression. 6-19
6.4.3 VALUE OF LOGICAL FACTORS, TERMS, AND EXPRESSIONS . . 6-19
6.4.4 LOGICAL CONSTANT EXPRESSION. 6-20
6.5 PRECEDENCE OF OPERATORS 6-21
6.5.1 SUMMARY OF INTERPRETATION RULES. 6-21
6.6 EVALUATION OF EXPRESSIONS 6-22
6.6.1 EVALUATION OF OPERANDS. 6-24
6.6.2 ORDER OF EVALUATION OF FUNCTIONS. 6-24
6.6.3 INTEGRITY OF PARENTHESES. 6-24
6.6.4 RESTRICTIONS ON APPEARANCE OF ARRAY EXPRESSIONS . . . 6-25
6.6.5 EVALUATION OF ARITHMETIC EXPRESSIONS 6-25
6.6.6 EVALUATION OF CHARACTER EXPRESSIONS 6-27
6.6.7 EVALUATION OF RELATIONAL EXPRESSIONS 6-27
6.6.8 EVALUATION OF LOGICAL EXPRESSIONS 6-28
6.7 BOOLEAN EXPRESSIONS 6-28
6.7.1 BOOLEAN OPERANDS 6-28
6.7.1.1 Boolean Primary 6-28
6.7.1.2 Boolean Factor 6-29
6.7.1.3 Boolean Term 6-29
6.7.1.4 Boolean Disjunct 6-29
6.7.1.5 Boolean Expression 6-30
6.7.2 VALUE OF BOOLEAN FACTORS, TERMS, AND EXPRESSIONS . . 6-30
6.7.3 BOOLEAN CONSTANT EXPRESSION 6-32
6.8 CONSTANT EXPRESSIONS 6-32
6.9 BIT EXPRESSIONS 6-33
6.9.1 BIT OPERATORS. 6-33
6.9.2 FORM AND INTERPRETATION OF BIT EXPRESSIONS. 6-34
6.9.2.1 Bit Primaries. 6-34
6.9.2.2 Bit Factor. 6-35
6.9.2.3 Bit Term. 6-35
6.9.2.4 Bit Disjunct. 6-35
6.9.2.5 Bit Expression. 6-35
6.9.3 VALUE OF BIT FACTORS, TERMS, AND EXPRESSIONS 6-36
6.9.4 BIT CONSTANT EXPRESSION. 6-37

7.0 STATEMENT CLASSIFICATION 7-1
7.1 EXECUTABLE STATEMENTS 7-1
7.2 NONEXECUTABLE STATEMENTS 7-2

8.0 SPECIFICATION STATEMENTS 8-1
8.1 DIMENSION STATEMENT 8-1
8.2 EQUIVALENCE STATEMENT 8-2
8.2.1 FORM OF AN EQUIVALENCE STATEMENT 8-2
8.2.2 EQUIVALENCE ASSOCIATION. 8-2
8.2.3 EQUIVALENCE OF ENTITIES OF DIFFERENT TYPES 8-2
8.2.4 ARRAY NAMES AND ARRAY ELEMENT NAMES. 8-4
8.2.5 RESTRICTIONS ON EQUIVALENCE STATEMENTS. 8-4
8.3 COMMON STATEMENT 8-4
8.3.1 FORM OF A COMMON STATEMENT. 8-5
8.3.2 COMMON BLOCK STORAGE SEQUENCE. 8-6
8.3.3 SIZE OF A COMMON BLOCK. 8-6
8.3.4 COMMON ASSOCIATION. 8-6
8.3.5 DIFFERENCES BETWEEN NAMED COMMON AND BLANK COMMON . . 8-6
8.3.6 RESTRICTIONS ON COMMON AND EQUIVALENCE. 8-7

TABLE OF CONTENTS

8.4 TYPE-STATEMENTS 8-7
8.4.1 BIT, BOOLEAN, AND ARITHMETIC TYPE-STATEMENTS 8-7
8.4.2 CHARACTER TYPE-STATEMENT. 8-8
8.5 IMPLICIT STATEMENT 8-9
8.6 PARAMETER STATEMENT 8-10
8.7 EXTERNAL STATEMENT 8-11
8.8 INTRINSIC STATEMENT 8-12
8.9 SAVE STATEMENT 8-13
8.10 ROWWISE STATEMENT 8-14
8.11 PROCEDURE INTERFACE INFORMATION 8-14
8.12 VIRTUAL STATEMENT 8-15

9.0 DATA STATEMENT 9-1
9.1 FORM OF A DATA STATEMENT 9-1
9.2 DATA STATEMENT RESTRICTIONS 9-1
9.3 IMPLIED-DO IN A DATA STATEMENT 9-3
9.4 CHARACTER CONSTANT IN A DATA STATEMENT 9-3

10.0 ASSIGNMENT STATEMENTS 10-1
10.1 ARITHMETIC ASSIGNMENT STATEMENT 10-1
10.2 LOGICAL ASSIGNMENT STATEMENT 10-2
10.3 STATEMENT LABEL ASSIGNMENT (ASSIGN) STATEMENT 10-2
10.4 CHARACTER ASSIGNMENT STATEMENT 10-3
10.5 MULTIPLE ASSIGNMENT STATEMENT 10-4
10.6 BOOLEAN ASSIGNMENT STATEMENT 10-4
10.7 BIT ASSIGNMENT STATEMENT 10-5
10.8 ARRAY ASSIGNMENT STATEMENTS 10-5
10.8.1 IDENTIFY STATEMENT 10-5
10.8.2 FORALL STATEMENT 10-8

11.0 CONTROL STATEMENTS 11-1
11.1 UNCONDITIONAL GO TO STATEMENT 11-2
11.2 COMPUTED GO TO STATEMENT 11-2
11.3 ASSIGNED GO TO STATEMENT 11-2
11.4 ARITHMETIC IF STATEMENT 11-3
11.5 LOGICAL IF STATEMENT 11-3
11.6 BLOCK IF STATEMENT 11-4
11.6.1 IF-LEVEL. 11-4
11.6.2 IF-BLOCK. 11-4
11.6.3 EXECUTION OF A BLOCK IF STATEMENT. 11-4
11.7 ELSE IF STATEMENT 11-5
11.7.1 ELSE IF-BLOCK. 11-5
11.7.2 EXECUTION OF AN ELSE IF STATEMENT. 11-5
11.8 ELSE STATEMENT 11-5
11.8.1 ELSE-BLOCK. 11-5
11.8.2 EXECUTION OF AN ELSE STATEMENT. 11-6
11.9 END IF STATEMENT 11-6
11.10 DO STATEMENT 11-6
11.10.1 RANGE OF A DO-LOOP. 11-7
11.10.2 ACTIVE AND INACTIVE DO-LOOPS. 11-7
11.10.3 EXECUTING A DO STATEMENT. 11-8
11.10.4 LOOP CONTROL PROCESSING. 11-8
11.10.5 EXECUTION OF THE RANGE. 11-9
11.10.6 TERMINAL STATEMENT EXECUTION. 11-9
11.10.7 INCREMENTATION PROCESSING. 11-9

TABLE OF CONTENTS

11.10.8 TRANSFER INTO THE RANGE OF A DO-LOOP. 11-10
11.11 CONTINUE STATEMENT 11-10
11.12 STOP STATEMENT 11-10
11.13 PAUSE STATEMENT 11-10
11.14 END STATEMENT 11-11
11.15 LOGICAL WHERE STATEMENT 11-11
11.16 BLOCK WHERE STATEMENT 11-12
11.16.1 WHERE-LEVEL 11-12
11.16.2 WHERE-BLOCK 11-12
11.16.3 EXECUTION OF A BLOCK WHERE STATEMENT 11-12
11.17 OTHERWISE STATEMENT 11-13
11.17.1 OTHERWISE-BLOCK 11-13
11.17.2 EXECUTION OF AN OTHERWISE STATEMENT 11-13
11.18 END WHERE STATEMENT 11-14

12.0 ARRAY STORAGE ALLOCATION 12-1
12.1 ALLOCATE STATEMENT 12-1
12.2 FREE STATEMENT 12-2

13.0 INPUT/OUTPUT STATEMENTS 13-1
13.1 RECORDS 13-2
13.1.1 FORMATTED RECORD. 13-2
13.1.2 UNFORMATTED RECORD 13-2
13.1.3 ENDFILE RECORD. 13-2
13.2 FILES 13-3
13.2.1 FILE EXISTENCE. 13-3
13.2.2 FILE PROPERTIES 13-3
13.2.3 FILE POSITION. 13-3
13.2.4 FILE ACCESS. 13-4
13.2.4.1 Sequential Access. 13-4
13.2.4.2 Direct Access. 13-5
13.2.5 INTERNAL FILES 13-6
13.2.5.1 Standard Internal File Properties. 13-6
13.2.5.2 Standard Internal File Restrictions. 13-7
13.2.5.3 Extended Internal File Properties 13-7
13.2.5.4 Extended Internal File Restrictions 13-8
13.3 UNITS 13-8
13.3.1 UNIT EXISTENCE. 13-8
13.3.2 CONNECTION OF A UNIT. 13-8
13.3.3 UNIT SPECIFIER AND IDENTIFIER. 13-9
13.4 FORMAT SPECIFIER AND IDENTIFIER 13-11
13.5 RECORD SPECIFIER 13-11
13.6 ERROR AND END-OF-FILE CONDITIONS 13-12
13.7 I/O STATUS, ERROR, AND END-OF-FILE SPECIFIERS 13-12
13.7.1 ERROR SPECIFIER. 13-13
13.7.2 END-OF-FILE SPECIFIER. 13-13
13.8 READ, WRITE, PRINT, AND PUNCH STATEMENTS 13-14
13.8.1 CONTROL INFORMATION LIST. 13-15
13.8.2 INPUT/OUTPUT LIST 13-16
13.8.2.1 Input List Items. 13-16
13.8.2.2 Output List Items. 13-16
13.8.2.3 Implied-DO List. 13-17
13.9 EXECUTION OF A DATA TRANSFER INPUT/OUTPUT STATEMENT . . . 13-17
13.9.1 DIRECTION OF DATA TRANSFER 13-18
13.9.2 IDENTIFYING A UNIT 13-18

TABLE OF CONTENTS

13.9.3 ESTABLISHING A FORMAT 13-18
13.9.4 FILE POSITION PRIOR TO DATA TRANSFER 13-19
13.9.4.1 Sequential Access 13-19
13.9.4.2 Direct Access 13-19
13.9.5 DATA TRANSFER 13-19
13.9.5.1 Unformatted Data Transfer 13-20
13.9.5.2 Formatted Data Transfer 13-21
13.9.5.2.1 USING A FORMAT SPECIFICATION 13-21
13.9.5.2.2 LIST-DIRECTED FORMATTING 13-21
13.9.5.2.3 PRINTING OF FORMATTED RECORDS 13-21
13.9.6 FILE POSITION AFTER DATA TRANSFER 13-22
13.9.7 INPUT/OUTPUT STATUS SPECIFIER DEFINITION 13-22
13.10 AUXILIARY INPUT/OUTPUT STATEMENTS 13-22
13.10.1 OPEN STATEMENT 13-22
13.10.1.1 OPEN of a Connected Unit. 13-26
13.10.2 CLOSE STATEMENT. 13-26
13.10.2.1 Implicit Close at Termination of Execution. . 13-28
13.10.3 INQUIRE STATEMENT. 13-28
13.10.3.1 INQUIRE by File. 13-28
13.10.3.2 INQUIRE by Unit. 13-28
13.10.3.3 Inquiry Specifiers. 13-29
13.10.4 FILE POSITIONING STATEMENTS. 13-33
13.10.4.1 BACKSPACE Statement. 13-34
13.10.4.2 ENDFILE Statement. 13-34
13.10.4.3 REWIND Statement. 13-34
13.11 RESTRICTIONS ON FUNCTION REFERENCES AND LIST ITEMS . . 13-34
13.12 RESTRICTION ON INPUT/OUTPUT STATEMENTS 13-35
13.13 NAMELIST INPUT/OUTPUT 13-35
13.13.1 NAMELIST STATEMENT 13-35
13.13.2 NAMELIST DATA TRANSFER 13-36
13.14 ENCODE AND DECODE STATEMENTS 13-36
13.15 BUFFER IN AND BUFFER OUT STATEMENTS 13-38

14.0 FORMAT SPECIFICATION 14-1
14.1 FORMAT SPECIFICATION METHODS 14-1
14.1.1 FORMAT STATEMENT. 14-1
14.1.2 CHARACTER FORMAT SPECIFICATION. 14-1
14.2 FORM OF A FORMAT SPECIFICATION 14-2
14.2.1 EDIT DESCRIPTORS 14-3
14.3 INTERACTION BETWEEN INPUT/OUTPUT LIST AND FORMAT 14-4
14.4 POSITIONING BY FORMAT CONTROL 14-5
14.5 EDITING 14-5
14.5.1 APOSTROPHE AND QUOTE EDITING 14-6
14.5.2 H EDITING 14-6
14.5.3 POSITIONAL EDITING. 14-6
14.5.3.1 T, TL, and TR Editing. 14-7
14.5.3.2 X Editing. 14-7
14.5.4 SLASH EDITING. 14-7
14.5.5 COLON EDITING. 14-8
14.5.6 S, SP, AND SS EDITING. 14-8
14.5.7 P EDITING. 14-8
14.5.7.1 Scale Factor. 14-8
14.5.8 BN AND BZ EDITING. 14-9
14.5.9 NUMERIC EDITING 14-9
14.5.9.1 Integer Editing 14-10

TABLE OF CONTENTS

14.5.9.2 Real, Double - , and Half - Precision Editing . 14-11
14.5.9.2.1 F EDITING 14-11
14.5.9.2.2 E AND D EDITING 14-12
14.5.9.2.3 G EDITING 14-13
14.5.9.2.4 COMPLEX EDITING 14-13
14.5.10 L EDITING 14-14
14.5.11 A EDITING 14-14
14.5.11.1 A Editing of Character Data 14-14
14.5.11.2 A Editing of Noncharacter Data 14-14
14.5.12 PROCESSOR-DEPENDENT EDITING 14-15
14.5.12.1 R Editing 14-15
14.5.12.2 O Editing 14-16
14.5.12.3 Z Editing 14-17
14.5.13 B EDITING 14-18
14.6 LIST-DIRECTED FORMATTING 14-19
14.6.1 LIST-DIRECTED INPUT 14-19
14.6.2 LIST-DIRECTED OUTPUT 14-21
14.7 NAMELIST FORMATTING 14-22
14.7.1 NAMELIST INPUT 14-24
14.7.2 NAMELIST OUTPUT 14-26

15.0 MAIN PROGRAM 15-1
15.1 PROGRAM STATEMENT 15-1
15.2 MAIN PROGRAM RESTRICTIONS 15-1

16.0 FUNCTIONS AND SUBROUTINES 16-1
16.1 CATEGORIES OF FUNCTIONS AND SUBROUTINES 16-1
16.1.1 PROCEDURES. 16-1
16.1.2 EXTERNAL FUNCTIONS. 16-1
16.1.3 SUBROUTINES. 16-1
16.1.4 DUMMY PROCEDURE. 16-1
16.2 REFERENCING A FUNCTION 16-1
16.2.1 FORM OF A FUNCTION REFERENCE. 16-2
16.2.2 EXECUTION OF A FUNCTION REFERENCE. 16-2
16.3 INTRINSIC FUNCTIONS 16-3
16.3.1 SPECIFIC NAMES AND GENERIC NAMES 16-3
16.3.2 REFERENCING AN INTRINSIC FUNCTION. 16-4
16.3.3 INTRINSIC FUNCTION ARGUMENTS AND RESULTS 16-4
16.3.4 INTRINSIC FUNCTION RESTRICTIONS. 16-5
16.3.5 ARRAY REDUCTION INTRINSIC FUNCTIONS 16-5
16.3.5.1 Array Reduction to a Scalar 16-5
16.3.5.2 Array Reduction Along a Dimension 16-5
16.4 STATEMENT FUNCTION 16-6
16.4.1 FORM OF A STATEMENT FUNCTION STATEMENT. 16-6
16.4.2 REFERENCING A STATEMENT FUNCTION. 16-7
16.4.3 STATEMENT FUNCTION RESTRICTIONS. 16-8
16.5 EXTERNAL FUNCTIONS 16-9
16.5.1 FUNCTION SUBPROGRAM AND FUNCTION STATEMENT 16-9
16.5.2 REFERENCING AN EXTERNAL FUNCTION. 16-10
16.5.2.1 Execution of an External Function Reference. . 16-10
16.5.2.2 Actual Arguments for an External Function. . . 16-10
16.5.3 FUNCTION SUBPROGRAM RESTRICTIONS 16-11
16.5.4 USER ARRAY-VALUED FUNCTION DECLARATION 16-12
16.5.4.1 Shape of a User Array-valued Function Result . . 16-12
16.5.4.2 User Array-Valued Function Name Usage 16-12

TABLE OF CONTENTS

16.5.4.3 User Array-Valued Function Restrictions 16-13
16.6 SUBROUTINES 16-13
16.6.1 SUBROUTINE SUBPROGRAM AND SUBROUTINE STATEMENT. . . 16-13
16.6.2 SUBROUTINE REFERENCE. 16-13
16.6.2.1 Form of a CALL Statement. 16-13
16.6.2.2 Execution of a CALL Statement. 16-14
16.6.2.3 Actual Arguments for a Subroutine. 16-14
16.6.3 SUBROUTINE SUBPROGRAM RESTRICTIONS. 16-15
16.7 ENTRY STATEMENT 16-15
16.7.1 FORM OF AN ENTRY STATEMENT. 16-16
16.7.2 REFERENCING AN EXTERNAL PROCEDURE BY AN ENTRY NAME . 16-16
16.7.3 ENTRY ASSOCIATION. 16-17
16.7.4 ENTRY STATEMENT RESTRICTIONS. 16-17
16.8 RETURN STATEMENT 16-18
16.8.1 FORM OF A RETURN STATEMENT. 16-18
16.8.2 EXECUTION OF A RETURN STATEMENT. 16-18
16.8.3 ALTERNATE RETURN. 16-19
16.8.4 DEFINITION STATUS. 16-19
16.9 ARGUMENTS AND COMMON BLOCKS 16-19
16.9.1 DUMMY ARGUMENTS. 16-20
16.9.2 ACTUAL ARGUMENTS 16-20
16.9.3 ASSOCIATION OF DUMMY AND ACTUAL ARGUMENTS. 16-21
16.9.3.1 Length of Character Dummy and Actual Arguments . 16-22
16.9.3.2 Variables as Dummy Arguments. 16-23
16.9.3.3 Arrays as Dummy Arguments 16-23
16.9.3.4 Procedures as Dummy Arguments. 16-25
16.9.3.5 Asterisks as Dummy Arguments. 16-25
16.9.3.6 Restrictions on Association of Entities. 16-26
16.9.4 COMMON BLOCKS. 16-26
16.10 TABLE 5 INTRINSIC FUNCTIONS 16-28
16.10.1 RESTRICTIONS ON RANGE OF ARGUMENTS AND RESULTS. . 16-50
16.11 PROCESSOR-SUPPLIED FUNCTIONS 16-51
16.11.1 DATE 16-51
16.11.2 TIME 16-52
16.11.3 SECOND 16-52
16.11.4 IOCLAS 16-52
16.11.5 NUMERR 16-53
16.11.6 UNIT 16-53
16.12 PROCESSOR-SUPPLIED SUBROUTINES 16-53
16.12.1 REMARK 16-54
16.12.2 CONNEC 16-54
16.12.3 DISCON 16-54
16.12.4 LIMERR 16-55
16.12.5 RANSET 16-55
16.12.6 RANGET 16-56

17.0 BLOCK DATA SUBPROGRAM 17-1
17.1 BLOCK DATA STATEMENT 17-1
17.2 BLOCK DATA SUBPROGRAM RESTRICTIONS 17-1

18.0 ASSOCIATION AND DEFINITION 18-1
18.1 STORAGE AND ASSOCIATION 18-1
18.1.1 STORAGE SEQUENCE. 18-1
18.1.2 ASSOCIATION OF STORAGE SEQUENCES 18-1
18.1.3 ASSOCIATION OF ENTITIES. 18-2

TABLE OF CONTENTS

18.2 EVENTS THAT CAUSE ENTITIES TO BECOME DEFINED 18-3
18.3 EVENTS THAT CAUSE ENTITIES TO BECOME UNDEFINED 18-5

19.0 SCOPE AND CLASSES OF SYMBOLIC NAMES 19-1
19.1 SCOPE OF SYMBOLIC NAMES 19-1
19.1.1 GLOBAL ENTITIES. 19-1
19.1.1.1 Classes of Global Entities. 19-1
19.1.2 LOCAL ENTITIES. 19-2
19.1.2.1 Classes of Local Entities. 19-2
19.2 CLASSES OF SYMBOLIC NAMES 19-2
19.2.1 COMMON BLOCK 19-3
19.2.2 EXTERNAL FUNCTION 19-3
19.2.3 SUBROUTINE 19-3
19.2.4 MAIN PROGRAM 19-4
19.2.5 BLOCK DATA SUBPROGRAM 19-4
19.2.6 ARRAY 19-4
19.2.7 VARIABLE 19-4
19.2.8 CONSTANT 19-5
19.2.9 STATEMENT FUNCTION 19-5
19.2.10 INTRINSIC FUNCTION 19-5
19.2.11 DUMMY PROCEDURE 19-6
19.2.12 NAMELIST GROUP NAME 19-6

A
A

Table of Contents

A
A