X3J3 / S8
April, 1986

American National Standard
for Information Systems
Programming Language

Fortran

S8 (X3.9-198x)
Revision of X3.9-1978

Secretariat: Computer and Business Equipment Manufacturers Association

Draft S8, Version 99
Submitted to X3 by X3J3, Americal National Standards Institute, Inc.

FOREWORD

American National Standard Language Fortran, X3.9-198x, specifies the form and establishes the interpretation of programs expressed in the Fortran Language. It consists of the specification of the language Fortran. No subsets are specified in this standard. The previous standard, commonly known as "Fortran 77", is entirely contained within this standard, known as "Fortran 8x". Any standard-conforming Fortran 77 program is intended to be a standard-conforming Fortran 8x program. New Fortran 8x features can be compatibly incorporated into such programs, with any exceptions clearly indicated in the text of this standard.

This document is released to SPARC, a subcommittee of X3, the American National Standards Committee for Information Processing Systems, operating under the procedures of the American National Standards Institute. The Computer and Business Equipment Manufacturers Association holds the secretariat. The purpose of this release is to submit the document for compliance review to SPARC and for preliminary information to X3 in anticipation of an X3 favorable vote to process the draft as an American National Standard.

Appendix A describes a "Fortran Family of Standards" as well as the philosophy used in partitioning the Fortran Language into new or incremental features, primary features, and obsolete or decremental features.

Since the publication of Fortran 77 (April 1978), the technical committee, X3J3, has been developing the draft revision. The central philosophy has been to modernize Fortran so that it may continue its long history as a scientific and engineering programming language.

The membership of the committee since that time is listed in the following section. Administration of X3J3 has been undertaken by a "Steering Committee" and the technical development has been carried out by subgroups, whose work is reviewed by the full committee. During the period of development of the draft Fortran standard, many persons assumed important roles of leadership. Their contributions are mentioned in the following section. At the present time, the membership consists of 40 members.

STEERING COMMITTEE

Jeanne Adams, Chair
Jerry Wagener, Acting Vice-Chair
Walt Brainerd, Director, Technical Work
Lloyd Campbell, Editor
Jeanne Martin, Secretary
Neldon Marshall, Librarian
Andrew Johnson, Interpretations
Jim Matheny, Vocabulary Representative

SUBGROUP HEADS

Dick Hendrickson
Kurt Hirchert
Jim Matheny
Rich Ragan
Andrew Johnson

The international community of Fortran experts has been very helpful in reviewing the development of this draft standard. At the most recent meeting of Working Group 5, Subcommittee 22 of Technical Committee 97 on Information Processing Systems, a resolution was passed that the work is "in general representative of the needs of the Fortran community worldwide...."

Subcommittee X3J3 on Fortran developed this standard. Those who contributed to the work of the subcommittee were:

Jeanne C. Adams, Chair
Jerrold L. Wagener, Acting Vice-Chair
Martin N. Greenfield, Vice-Chair (1972-1985)
Walter S. Brainerd, Director, Technical Work*
Lloyd W. Campbell, Editor*
Jeanne T. Martin, Secretary*
Loren P. Meissner, Secretary (1978-1982)
Jeanne T. Martin, Acting International Representative
Frances E. Holberton, International Representative (1978-1982)
Neldon H. Marshall, Librarian*

Cornelis G. F. Ampt
Stuart L. Anderson
Charles Arnold
Graham Barber
Gloria M. Bauer*
Valerie G. Bowe
Joanne Brixius
Neil Brutman
Larry Bumgarner
Carl D. Burch
Winfried A. Burke*
John H. Carman
T. C. Chao
Nancy Cheng
Joel Clinkenbeard
Joe Cointment
Ted Crowley
Chela Diaz de Villegas
David C. Dillon
Joe L. Dowdell
John T. Engle
Stuart I. Feldman
Murray F. Freeman
Daniel A. Gallagher
Gary L. Graunke
Stephen R. Greenwood
Richard B. Grove*
Kevin W. Harris
Richard A. Hendrickson*
Dean A. Herington*
Kurt W. Hirchert*
Steve K. Hue
E. Andrew Johnson*
Gregory Johnson
Peter N. Karculias
Leslie M. Klein
Wilfried Kneis
Werner Koblitz
George T. Komorowski
Joe A. Korty
Dorothy E. Lang
John E. Lauer*
Kay Leonard
Donald L. Loe
Warren E. Loper
Bruce A. Martin*
Alex L. Marusak
James H. Matheny*
John Mayer
Edward H. McCall
Michael Metcalf
Geoff Millard
Robert M. Miller
Leonard J. Moss
David T. Muxworthy
Linda J. O'Gara
Rod R. Oldehoeft
John P. Olson*
Rex L. Page*
George Paul
Daniel Pearl
Odd Pettersen
Ivor R. Philips
Bruce W. Puerling*
Richard R. Ragan*
John K. Reid
Steven M. Rowan
Werner Schenk*
Lawrie J. Schonfelder
Rick N. Schubert
John C. Schwebel
Richard Shepardson
Richard W. Signor*
Brian T. Smith*
Jan A. M. Snoek
Hieronymus Sobiesiak
Ken Sperka
Bruce Stowell
Sylvia Sund
Mario Surdi
Richard C. Swift
Brian L. Thompson
Robert B. Upshaw*
Richard W. Weaver
George E. Weekly
Bruce Weinman
Everett H. Whitley
Gunter Wiesner
Edward J. Wilkens
Alan Wilson

*Subgroup Head

TABLE OF CONTENTS

1 INTRODUCTION

1.1 Purpose. This standard specifies the form and establishes the interpretation of programs expressed in the Fortran language. The purpose of this standard is to promote portability, reliability, maintainability, and efficient execution of Fortran programs for use on a variety of computing systems. This standard is an upward compatible extension to the preceding Fortran standard, X3.9-1978, informally referred to as Fortran 77. Any standard-conforming Fortran 77 program is standard conforming under this standard, with the same interpretation; however, see Section 1.4 regarding intrinsic procedures.

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

1.3 Scope. This standard specifies the bounds of the Fortran language by identifying both those items included and those items excluded.

1.3.1 Inclusions. This standard specifies:

(1) The forms that a program written in the Fortran language may take

(2) The rules for interpreting the meaning of a program and its data

(3) The form of the input data to be processed by such a program

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

1.3.2 Exclusions. This standard does not specify:

(1) The mechanism by which programs are transformed for use on computers

(2) The operations required for setup and control of the use of programs on computers

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

(4) The program and processor behavior 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 computing system or the capability of a particular processor

(6) The physical properties of the representation of quantities and the method of rounding of numeric values on a particular processor

(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 refer to permissible forms and relationships for **standard-conforming programs** rather than for processors. The optional output forms produced by a processor, which are not under the control of a program, are an example of an exception. The requirements, prohibitions, and options for a standard-conforming processor must be inferred from those given for programs.

An executable program (2.2.1) 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 (2.2) 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 procedures even though this could cause a conflict with the name of an external or internal procedure in a standard-conforming program. If such a conflict occurs and involves the name of an external procedure, the processor is permitted to use the intrinsic procedure unless the name appears in an EXTERNAL statement within the program unit. A standard-conforming program must not use nonstandard intrinsic procedures that have been added by the processor.

This standard has more intrinsic procedures than did Fortran 77. Therefore, a standard-conforming Fortran 77 program may have a different interpretation under this standard if it invokes a procedure having the same name as one of the new standard intrinsic procedures, unless that procedure is specified in an EXTERNAL statement as recommended for nonintrinsic functions in the appendix to the Fortran 77 standard.

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. For example, a standard-conforming processor may allow a nonstandard data type such as INTEGER*2.

Because a standard-conforming program may place demands on a 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.

1.5.1 Syntax Rules.

Syntax rules are used to help describe the form that Fortran statements and constructs may take. These syntax rules are a variation of Backus-Naur form (BNF) in which:

(1) Characters from the Fortran character set are to be written as shown, except where otherwise noted.

(2) Lower case italicized letters and words (often hyphenated and abbreviated) represent general syntactic classes for which specific syntactic entities must be substituted in actual statements.

Some common abbreviations used in syntactic terms are:

stmt	for	statement	*attr*	for	attribute
expr	for	expression	*decl*	for	declaration
spec	for	specifier	*def*	for	definition
int	for	integer	*desc*	for	descriptor
arg	for	argument	*op*	for	operator

(3) The syntactic metasymbols used are:

is	introduces a syntactic class definition
or	introduces a syntactic class alternative
[ ]	encloses an optional item
[ ]...	encloses an optionally repeated item which may occur zero or more times
□	continues a syntax rule

(4) Each syntax rule is given a unique identifying number of the form R*snn*, where *s* is a one or two digit section number and *nn* is a sequence number within that section. The syntax rules are distributed as appropriate throughout the text, and may be referenced by number as needed.

(5) The syntax rules are not a complete and accurate syntax description of Fortran, and cannot be used to generate automatically a Fortran parser; where a syntax rule is incomplete, it is accompanied by an informal description of the corresponding constraint.

(6) Deprecated features are shown in a distinguishing type font. This is an example of the font used for deprecated features.

An example of the use of syntax rules is:

int-constant	**is** *digit* [[*underscore*] *digit*]...

The following forms are examples of forms for an integer constant allowed by the above rule:

digit
digit digit
digit underscore digit digit digit
digit digit underscore digit digit digit underscore digit digit digit

When specific entities are substituted for *digit* and *underscore* actual integer constants might be:

4
67
1_999
10_243_852

1.5.2 Assumed Syntax Rules. To minimize the number of additional syntax rules and convey appropriate constraint information, the following rules are assumed unless explicitly overridden. The letters "xyz" stand for any legal syntactic class phrase:

xyz-list	**is** *xyz* [, *xyz*]...
xyz-name	**is** *symbolic-name*
xyz-symbolic-constant	**is** *symbolic-name*
xyz-expr	**is** *expr*
xyz-variable	**is** *variable*
int-xyz	**is** *xyz*
char-xyz	**is** *xyz*
derived-type-xyz	**is** *xyz*
scalar-xyz	**is** *xyz*
array-xyz	**is** *xyz*

1.5.3 Syntax Conventions and Characteristics.

(1) Any syntactic class name ending in "-stmt" follows the source form statement rules: it may be labeled and must be delimited by end-of-line or semicolon. Conversely, everything considered to be a source form statement is given a "-stmt" ending in the syntax rules.

(2) Statement ordering is rigorously described in the definition of *program-unit-body* (R210). Expression hierarchy is rigorously described in the definition of *expr* (R715).

(3) The term "type parameter" applies to a data type parameter, with "*type-param-name*" used for the dummy parameter and "*type-param-spec*" (R503) used for the actual parameter, including the optional keyword. The part without the keyword is called "*type-param-value*" (R504). These terms parallel the use of "*dummy-arg-name*", "*actual-arg-spec*" (R1212) and "*actual-arg*" (R1214), respectively, for procedure arguments. In the case of intrinsic type parameters, CHARACTER length *type-param-spec* is called "*length-param-spec*" (R508), and REAL precision and exponent range *type-param-spec*s are called "*precision-param-spec*" (R507) and "*exp-range-param-spec*" (R507), respectively.

(4) The suffix "*-spec*" is used consistently for specifiers, such as keyword type parameters, keyword actual arguments, and input/output statement specifiers. It also is used for type declaration attribute specifications (e.g., "*array-spec*"), and in a few other ad hoc cases.

(5) When reference is made to a parameter, including the surrounding parentheses, the term "selector" is used. See, for example, "*length-selector*" (R508), "*precision-selector*" (R409, R507), "*array-selector*" (R605), and "*case-selector*" (R813).

(6) The term "*subscript*" (e.g., R611 and R614) is used consistently in array definitions.

1.5.4 Text Conventions. In the descriptive text, the normal English word equivalent of a BNF syntactic term is usually used. Specific statements are identified in the text by the uppercase keyword, e.g., "END statement". Boldface words are also used in the text where they are first defined with a specialized meaning.

1.6 Deprecated Features, Core Conformance. Since it first became available in 1957, Fortran has undergone several generations of development and evolution. This is the third Fortran standard, the first having appeared in 1966 and the second in 1978. Fortran has changed significantly during this period, and some of its newer features are more effective for reliable software production than certain earlier features. Therefore, some elements of the language are identified in this standard as **deprecated features** and are intended to be removed from the next version of the Fortran standard. It is emphasized, however, that the deprecated features are part of this version of the standard Fortran language. This identification of deprecated features permits Fortran users to minimize the impact of removal as follows:

(1) By using new features in new programs

(2) By replacing deprecated features with more effective features as existing programs are enhanced.

(3) By planned conversion for those remaining programs that must execute on standard-conforming processors for the next version of the Fortran standard.

The deprecated features are identified by means of a distinguishing type font (see Section 1.5 for an illustration of this font). Major deprecated features also are listed in Section 15 with a list of possible alternative features in Appendix B.

The set of language facilities in this standard that are not identified as deprecated features are referred to collectively as the **core**, which is a complete language. A standard-conforming executable program that does not contain any deprecated features is a **core-conforming program**. A standard-conforming program unit that does not contain any deprecated features is a **core-conforming program unit**.

1.7 Modules. This standard provides facilities that encourage the design and use of modular and reusable software. Data and procedure definitions may be organized into nonexecutable program units, called modules, and made available to any other program unit. In addition to global data and procedure library facilities, modules provide a mechanism for defining data abstractions and certain language extensions.

An **intrinsic module** is a module definition included with this standard. In addition, a module may be standardized as a separate collateral standard. A **standard module** must be core conforming. Operators defined in the module must not have the potential to alter the meaning of any core-conforming intrinsic operation.

2 FORTRAN TERMS AND CONCEPTS

2.1 High Level Syntax. The **high level syntax** introduces the terms associated with program units and other Fortran concepts above the construct, statement, and expression levels and illustrates their relationships. The syntax rule notation is described in 1.5.

R201	*executable-program*	**is** *external-program-unit* [*external-program-unit*]...

Constraint: An *executable-program* must contain exactly one *main-program program-unit*.

R202	*external-program-unit*	**is** *main-program*
		or *external-subprogram*
R203	*main-program*	**is** [*program-stmt*] *program-unit-body* *end-program-stmt*
R204	*external-subprogram*	**is** *external-proc-subprogram*
		or *module-subprogram*
		or *block-data-subprogram*
R205	*external-proc-subprogram*	**is** *procedure-subprogram*
R206	*procedure-subprogram*	**is** *function-subprogram*
		or *subroutine-subprogram*
R207	*function-subprogram*	**is** *function-stmt* *program-unit-body* *end-function-stmt*
R208	*subroutine-subprogram*	**is** *subroutine-stmt* *program-unit-body* *end-subroutine-stmt*
R209	*module-subprogram*	**is** *module-stmt* *program-unit-body* *end-module-stmt*

Constraint: A module *program-unit-body* must not contain an *execution-part*.

R210	*block-data-subprogram*	is *block-data-stmt* *program-unit-body* *end-block-data-stmt*

Constraint: A *block-data-subprogram program-unit-body* may contain only IMPLICIT, PARAMETER, type declaration, COMMON, DIMENSION, EQUIVALENCE, DATA, and SAVE statements.

R211	*program-unit-body*	**is** [*use-stmt*]... [*implicit-part*]... [*declaration-part*]... [*stmt-function-part*]... [*execution-part*]... [*contains-stmt* [*internal-proc-subprogram*]...]
R212	*implicit-part*	**is** *implicit-stmt*
		or *parameter-stmt*

or *format-stmt*
or *entry-stmt*

Constraint: The last *implicit-part*, if any, in a program unit body must be an *implicit-stmt*.

R213 *declaration-part* **is** *derived-type-def*
or *interface-block*
or *type-declaration-stmt*
or *specification-stmt*
or *parameter-stmt*
or *format-stmt*
or *entry-stmt*

R214 *stmt-function-part* **is** *format-stmt*
or *data-stmt*
or *entry-stmt*
or *stmt-function-stmt*

Constraint: The first *stmt-function-part*, if any, in a program unit body must be a *stmt-function-stmt*.

R215 *execution-part* **is** *executable-construct*
or *format-stmt*
or *data-stmt*
or *entry-stmt*

Constraint: The first *execution-part*, if any, in a program unit body must be an *executable-construct* or a DATA statement.

R216 *internal-procedure* **is** *proc-subprogram*

R217 *specification-stmt* **is** *access-stmt*
or *condition-stmt*
or *exponent-letter-stmt*
or *external-stmt*
or *initialize-stmt*
or *intent-stmt*
or *intrinsic-stmt*
or *optional-stmt*
or *range-stmt*
or *save-stmt*
or *common-stmt*
or *dimension-stmt*
or *equivalence-stmt*

Constraint: An *intent-stmt* or *optional-stmt* must not appear in a module or main program because they apply only to dummy arguments.

R218 *executable-construct* **is** *action-stmt*
or *case-construct*
or *do-construct*
or *enable-construct*
or *if-construct*
or *where-construct*

R219 *action-stmt* **is** *allocate-stmt*

or *assignment-stmt*
or *backspace-stmt*
or *call-stmt*
or *close-stmt*
or *continue-stmt*
or *cycle-stmt*
or *deallocate-stmt*
or *endfile-stmt*
or *exit-stmt*
or *forall-stmt*
or *goto-stmt*
or *identify-stmt*
or *if-stmt*
or *inquire-stmt*
or *open-stmt*
or *print-stmt*
or *read-stmt*
or *return-stmt*
or *rewind-stmt*
or *set-range-stmt*
or *signal-stmt*
or *stop-stmt*
or *where-stmt*
or *write-stmt*
or *arithmetic-if-stmt*
or *assign-stmt*
or *assigned-goto-stmt*
or *computed-goto-stmt*
or *pause-stmt*

Constraint: An *entry-stmt* or *return-stmt* must not appear in a main program; an *entry-stmt* must not appear in constructs.

2.2 Program Unit Concepts. Program units are the fundamental components of a Fortran program. A program unit may be a main program, procedure subprogram, module subprogram, or block data subprogram. A procedure subprogram may be a function subprogram or a subroutine subprogram. A module contains definitions that are to be made ???available to other program units. A block data subprogram is used only to specify initial values for named common block data objects. Each type of program unit is described in Section 11 or 12.

2.2.1 Executable Program. An **executable program** consists of exactly one main program and any number (including zero) of external subprograms. The set of subprograms in the executable program may include any combination of the different kinds of subprograms in any order.

2.2.2 Main Program. Execution of an executable program begins with the first executable construct of the **main program**. The main program is described in detail in ???Section 11.1.

any of the data objects accessible to the subroutine; a function subprogram may do this in addition to computing the function value.

Procedures are described further in Section 12.

2.2.3.1 External Procedure. An **external procedure** is a nonintrinsic procedure whose definition is not contained within another program unit. An external procedure may be invoked by the main program or any procedure of an executable program.

2.2.3.2 Internal Procedure. An **internal procedure** is a procedure whose definition is contained within another program unit. The containing program unit is called the **host** of the internal procedure. An internal procedure is local to its host in the sense that the internal procedure is accessible within the host but is not accessible outside the host except through explicit means that make local entities accessible outside their hosts. Any kind of program unit, except a block data subprogram, may host internal procedures, and an internal procedure may host other internal procedures.

2.2.3.3 Procedure Interface Block. The purpose of a **procedure interface block** is to describe to the invoking program the attributes and keyword names of dummy arguments and, if the procedure is a function, the attributes of function results. It specifies the number of arguments a procedure has, the data type of each argument, the optional argument keywords that may be used in invoking the procedure, and which arguments (if any) are optional. It may also specify the argument intents (IN, OUT, or INOUT). A procedure interface block may be placed in a program unit that invokes the procedure, or in a module to which the invoking procedure has access. Procedure interface blocks are described in Section 12.

2.2.4 Module. A **module** contains (or accesses from other modules) definitions that are to be made accessible to other external program units. These definitions include data object declarations, type definitions, internal procedure definitions, and procedure interface blocks. A module cannot contain any definitions or specifications that could not be placed in other external program units. The purpose of a module is to make the definitions it contains accessible to all other program units in an executable program that requests such accessibility. Modules are global to all of the program units in an executable program, and any set of program units in the executable program may request access to the definitions contained in a module. The facilities contained in a module are global to the executable program, with the accessibility of the global facilities controlled on a program unit by program unit basis. Modules are further described in Section 11.

2.3 Execution Concepts.

A program unit is a sequence of statements. Statements are classified as **executable statements** and **nonexecutable statements**. Fortran places restrictions upon the order in which statements may appear in a program unit, and allows certain executable statements to appear only in an executable construct.

2.3.1 Executable/Nonexecutable Statements. Program execution is a sequence, in time, of computational actions. An executable statement is an instruction to perform or control one or more of these actions. Thus, the executable statements of a program unit determine the computational behavior of the program unit. The executable statements are all of those that make up the syntactic class of *executable-part*, except for *format-stmt*, *data-stmt*, and *entry-stmt*.

Nonexecutable statements do not specify actions; they are used to configure the program environment in which computational actions take place. The nonexecutable statements are all those not classified as executable. All statements in a block data subprogram must be nonexecutable. A module may contain executable statements only within an internal procedure definition.

2.3.1 Executable/Nonexecutable Statements. Program execution is a sequence, in time, of computational actions. An executable statement is an instruction to perform or control one or more of these actions. Thus, the executable statements of a program unit determine the computational behavior of the program unit. The executable statements are all of those that make up the syntactic class of *executable-part*, except for *format-stmt*, *data-stmt*, and *entry-stmt*.

Nonexecutable statements do not specify actions; they are used to configure the program environment in which computational actions take place. The nonexecutable statements are all those not classified as executable. All statements in a block data subprogram must be nonexecutable. A module may contain executable statements only within an internal procedure definition.

2.3.2 Statement Order. The syntax rules of Section 2.1 specify the statement order within program units. Figure 2.1 illustrates statement ordering. Vertical lines delineate varieties of statements that may be interspersed and horizontal lines delineate varieties of statements that must not be interspersed. USE statements, if any, must appear immediately after the program unit heading and internal procedure definitions must follow a CONTAINS statement. Between USE statements and internal procedure definitions nonexecutable statements generally precede executable statements, though the FORMAT statement, DATA statement, and ENTRY statement may appear among the executable statements.

Figure 2.1. Constraints on Statement Ordering.

PROGRAM, FUNCTION, SUBROUTINE, MODULE, or BLOCK DATA Statement		
USE Statements		
FORMAT and ENTRY Statements	PARAMETER Statements	IMPLICIT Statements
		Derived Type Definitions Interface blocks Type Declaration Statements Specification Statements
	DATA Statements	Statement Function Statements
		Executable Statements
CONTAINS Statement		
Internal Procedure Definitions		
Program Unit END Statement		

2.3.3 The END Statement. The program unit END statement must appear only as the terminal statement of a program unit definition. The terminal statement of each program unit must be an END statement. In all cases, the keyword END is a complete and valid END statement. Variations allowed by each kind of program unit are included with the descriptions of the program units (Sections 11 and 12). In main programs and procedure subprograms, the END statement may be executed, and its execution terminates execution of the program unit (equivalent to a STOP statement in main programs and a RETURN statement in procedures). An END statement may be labeled and may be the target of a program branch.

2.3.4 Execution Sequence. The execution of a main program or procedure involves execution of its executable constructs. Upon invocation of a procedure, execution begins with the first executable construct appearing after the invoked entry point. With the following exceptions, the executable constructs are executed in the order in which they appear in the main program or procedure until a STOP, RETURN, or program unit END statement is executed. The exceptions are:

(1) Execution of a branching statement (8.2) changes the execution sequence. These statements explicitly specify a new starting place for the execution sequence, and are called explicit branches.

(2) IF constructs, CASE constructs, and DO constructs contain an internal statement structure and execution of these constructs involves implicit (i.e., automatic) internal branching. See Section 8 for the detailed semantics of each of these constructs.

(3) Execution of a SIGNAL statement, or any statement that causes a condition to be raised, results in a change in the execution sequence.

(4) Alternate return and END= and ERR= specifiers may result in a branch.

(5) Internal procedure definitions may precede the END statement of a program unit. The execution sequence skips all such definitions.

(6) Handler definitions may precede the END ENABLE statement of an ENABLE construct. The execution sequence skips such definitions.

2.4 Data Concepts.

Nonexecutable statements are used to define the characteristics of the data environment. This includes typing variables, declaring arrays, and defining new data types.

2.4.1 Data Type. A **data type** consists of a set of values, together with a way to denote these values and a collection of operations that interpret and manipulate the values. This central concept is described in 4.1.

There are two categories of data types: intrinsic types and derived types.

2.4.1.1 Intrinsic Type. An **intrinsic type** is one that is implicitly defined, along with operations, and is always ???available. The intrinsic types are INTEGER, REAL, COMPLEX, DOUBLE PRECISION, CHARACTER (of any length), LOGICAL, and BIT. The properties of intrinsic types are described in ???Section 4.3.

2.4.1.2 Derived Type. A **derived type** is a type definition containing components, which are intrinsic types or other derived types. Derived types have associated with them a small set of intrinsic operations: assignment with type agreement, comparison for equality, use as procedure arguments and function results, and input/output. If additional operations are needed for a derived type, they must be supplied as procedure definitions.

Intrinsic types are ???available to every program unit. A derived-type definition is local to the program unit in which it appears. In order to make such a definition ???available to other program units in an executable program, the derived type may be defined in a module. Program units using a module have access to the derived-type definitions it contains.

Derived types are described further in 4.4.

2.4.2 Data Value. Each intrinsic type has associated with it a set of intrinsic **values** that objects of that type may take. These values for each intrinsic type are described in ???Section 4.3. Because derived types are specified in terms of intrinsic types, the intrinsic values also determine the values that objects of a derived type may assume.

2.4.3 Data Entity. A **data entity** is an entity that has, or may have, a data value. A data entity is either a constant or a variable. In addition, it is either a scalar or an array. Expression values and function results are considered to be data entities, either scalar or array, and have some of the properties of constants (for example, as procedure arguments).

2.4.3.1 Data Object. A **data object** is a named datum or set of data of the same type and type parameters that has a symbolic name and may be referenced as a whole. It may be a simple variable or symbolic constant.

2.4.3.2 Subobjects. Portions of certain data objects may be referenced and defined independently of the other portions. These include portions of arrays (array elements and array sections), portions of character strings (substrings), and portions of structured objects (components). These subobjects are described in Section 6.

2.4.3.3 Constant. A **constant** is a data entity whose value must not change during execution of an executable program.

A constant with a symbolic name is called a **symbolic constant**. Symbolic constants and the means by which they are defined are described in Section 5. A constant without a symbolic name is called a **literal constant**.

2.4.4 Variable. A **variable** is a data object or subobject whose value can be defined and redefined during execution of an executable program. A data object explicitly declared as an array and not having the PARAMETER attribute is a variable (array variable name). A nonarray data object, declared explicitly or implicitly and not having the PARAMETER attribute is a variable (scalar variable name). In some cases, a portion of a variable may itself be a variable and may be assigned a value independently of the other portions. The following data objects are variables:

a scalar variable name	(a scalar object)
an array variable name	(an array object)
an array element	(a scalar object)
an array section	(an array object)
a derived type component	(either a scalar or array object)
a substring	(a scalar or an array object)

2.4.4.1 Scalar. A **scalar** is a datum that is not an array or an array section. Scalars include

scalar variables and constants
array elements
substring of a scalar
scalar components of derived type objects

Scalars may be of any intrinsic type or derived type.

2.4.4.2 Array. An **array** is a set of data, all of the same type and type parameters, whose individual elements are arranged in a rectangular pattern. An **array element** is one of the individual elements in the array.

An aray with a symbolic name has one subscript for each dimension of the pattern. The pattern may have dimensions up to seven, and any **extent** (size) in any dimension. The **rank** of the array is the number of dimensions, and its **size** is the total number of elements, which is equal to the product of the extents. Arrays may have zero size. The **shape** of an array is determined by its rank and its extent in each dimension; shape is a rank one array whose elements are the extents. The rank of a scalar is zero. All arrays must be declared, and the rank of an array is specified in its declaration. The rank of an array, once declared, is constant and the extents may be constant also. However, the extents may vary during execution for dummy argument arrays, automatic arrays, alias arrays, ranged arrays, and allocatable arrays.

Two arrays are said to be **conformable** if they have the same shape. A scalar is conformable with any array. Any operation defined for scalar objects may be applied to conformable objects. Such operations are performed **element wise** to produce a resultant array conformable with the array operands. Element-wise operation means corresponding elements of the operand arrays are involved in a "scalar-like" operation to produce the corresponding element in the result array, and all such element operations may be performed simultaneously.

A one-dimensional array of scalar constants may be reshaped into any allowable array shape.

Array objects may be of any intrinsic type or derived type and are described further in Section 6.

2.4.5 Storage. Many of the facilities of this standard make no assumptions about the physical storage characteristics of data objects. However, program units that include **storage association** dependent features (Section 14) must observe certain storage constraints.

There are two kinds of physical **storage units**: numeric and character. When used in a storage association context, scalar objects of type integer, default real, and logical each use a single numeric storage unit. When used in a storage association context, scalar objects of type double precision and default complex each use two contiguous numeric storage units. When used in a storage association context, each character in an object of type character uses one character storage unit and scalar character objects employ a contiguous set of such units. When used in a storage association context, array objects are assigned contiguous storage units of the appropriate type, in subscript order value (Section 6). For example, the storage order for a two-dimensional array is the first column followed by the second column, etc.

Objects having different kinds of storage units must not be storage associated. Nondefault precision type objects, derived type objects, and bit objects must not appear in a storage association context.

2.5 Fundamental Terms.

The following terms are defined here and used throughout this standard.

2.5.1 Symbolic Name. A symbolic name is used to identify a program constituent, such as a program unit, variable, symbolic constant, dummy argument, or a derived type name. The rules governing the construction of symbolic names are given in 3.2.

2.5.2 Keyword. The term **keyword** is used in two ways in this standard. The words that are always part of the syntax of a statement and may be used to identify the statement are **statement keywords**. Examples of this kind of keyword are: IF, READ, WHERE, and INTEGER. These keywords are not "reserved words"; that is, symbolic names with the same spellings are allowed.

Argument keywords are dummy argument names. Section 13 defines argument keywords for all of the intrinsic procedures. Argument keywords for nonintrinsic procedures may be made ???available outside the procedure definition by making the procedure interface explicit (Section 12).

2.5.3 Declaration. The term **declaration** refers to the specification of attributes for various program entities. Often this involves specifying the data type of a data object or specifying the shape of an array object.

2.5.4 Definition. The term **definition** is used in two ways. First, when a data object is given a valid value during program execution, it is said to become *defined*. This is often accomplished by execution of an assignment statement or input statement. Section 14 describes the ways in which data objects may become defined and undefined. The second use of the term definition is for the definition of derived types and procedures.

2.5.5 Reference. A **data object reference** is the appearance of the data object in a context requiring its value at that point during execution.

A **procedure reference** is the appearance of the procedure name in a context requiring execution of the procedure at that point.

The appearance of a data object or procedure name in an actual argument list does not constitute a reference to that data object or procedure unless such a reference is needed to complete the specification of the actual argument.

2.5.6 Association. An **association** exists if an entity may be identified by different names in the same program unit or by the same name or different names in different program units. The forms of association are described in 14.2.

2.5.7 Intrinsic. The term **intrinsic** applies to intrinsic data types, intrinsic procedures, intrinsic operators, and intrinsic conditions that are defined in this standard. These may be used in any program unit without further definition or specification.

2.5.8 Operator. An **operator** specifies a particular computation involving one (unary operator) or two (binary operator) data values (operands). Fortran contains a number of intrinsic operators (e.g., the arithmetic operators +, −, *, /, and ** with numeric operands and the logical operators .AND., .OR., etc. with logical operands). Additional operators may also be defined.

2.5.9 Handler. A **handler** is a sequence of statements with a HANDLE statement as the first statement and terminated with the statement before the next HANDLE statement or END ENABLE statement, whichever comes first. A handler may be executed when a specified condition is signaled.

2.5.10 Condition. A **condition** is signaled when it is inappropriate to continue the normal execution sequence. A condition may be signaled by the execution of a SIGNAL statement (8.1.5.1) or it may be signaled implicitly. See 8.1.5.4 for the list of intrinsic conditions.

3 LEXICAL ELEMENTS

This section describes the Fortran character set and the various lexical elements such as symbolic names and operators. This section also describes the rules for the forms that Fortran programs may take.

3.1 Fortran Character Set. The Fortran **character set** consists of twenty-six letters, ten digits, underscore, and twenty-three special characters.

R301	*character*	**is** *alphanumeric-character*
		or *special-character*
R302	*alphanumeric-character*	**is** *letter*
		or *digit*
		or *underscore*

3.1.1 Letters. The twenty-six **letters** are:

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

If a processor also permits lower-case letters, the lower-case letters are equivalent to upper-case letters in program units except in character constants, delimited character edit descriptors, and H edit descriptors.

3.1.2 Digits. The ten **digits** are:

0 1 2 3 4 5 6 7 8 9

When used in numeric constants, the digits are interpreted according to the decimal base number system.

3.1.3 Special Characters. The twenty-three **special characters** plus underscore, which is considered to be an alphanumeric character, are:

Character	Name of Character	Character	Name of Character
	Blank	:	Colon
=	Equals	!	Exclamation Point
+	Plus	"	Quotation Mark or Quote
−	Minus	%	Percent
*	Asterisk	&	Ampersand
/	Slash	;	Semicolon
(	Left Parenthesis	<	Less Than
)	Right Parenthesis	>	Greater Than
,	Comma	?	Question Mark
.	Decimal Point or Period	[	Left Bracket
$	Currency Symbol	]	Right Bracket
'	Apostrophe	_	Underline or Underscore

The special characters are used for operator symbols, bracketing, and various forms of separating and delimiting of other lexical elements. The special characters $ and ? have no specified use. The underscore (_) may be used as a significant character in symbolic names and as an insignificant character in numeric constants.

3.1.4 Character Graphics. Except for the currency symbol, the graphics used for the characters must be as given in 3.1.1, 3.1.2, and 3.1.3. However, the style of any graphic is not specified.

3.1.5 Collating Sequence. Each implementation defines a collating sequence for the character set. A **collating sequence** is a one-to-one mapping of the characters into the nonnegative integers such that each character corresponds to a different nonnegative integer. The intrinsic functions CHAR and ICHAR (see Section 13) provide conversions between the characters and the integers according to this mapping. Thus,

ICHAR (*character*)

returns the integer value of the specified character according to the collating sequence of the processor.

The only constraints on the collating sequence are:

(1) ICHAR('A') < ICHAR('B') < · · · < ICHAR('Z') for the twenty-six letters.

(2) ICHAR('0') < ICHAR('1') < · · · < ICHAR ('9') for the ten digits.

(3) ICHAR(blank) < ICHAR('0') < ICHAR('9') < ICHAR('A') or
ICHAR(blank) < ICHAR('A') < ICHAR('Z') < ICHAR('0')

(4) ICHAR('a') < ICHAR('b') < · · · < ICHAR('z'), if a processor supports lower case letters

Except for blank, there are no constraints on the location of the special characters and underscore in the collating sequence, nor is there any specified collating sequence relationship between the upper-case and lower-case letters.

Note that the intrinsic functions ACHAR and IACHAR provide conversions between the characters and the integers according to the mapping specified in ANS X3.4-1977 (ASCII).

3.2 Low-Level Syntax.

The **low-level syntax** describes the fundamental lexical elements of a program unit. These are sequences of characters and include keywords, symbolic names, constants, operators, labels, and delimiters.

3.2.1 Keywords. Keywords appear as upper-case words in the syntax rules in Sections 4 through 12.

Some keywords may be written as either two separate consecutive words (e.g., END IF) or a single word (e.g., ENDIF). These double keywords are: BLOCK DATA, DOUBLE PRECISION, ELSE IF, ELSE WHERE, END BLOCK DATA, END DO, END ENABLE, END FILE, END FUNCTION, END IF, END INTERFACE, END MODULE, END PROGRAM, END SELECT, END SUBROUTINE, END TYPE, END WHERE, EXPONENT LETTER, FOR ALL, GO TO, IMPLICIT NONE, IN OUT, and SELECT CASE.

3.2.2 Symbolic Names. **Symbolic names** are names for various entities such as variables, program units, dummy arguments, symbolic constants, and derived types.

R303 *symbolic-name* **is** *letter* [*alphanumeric-character*]...

Constraint: The maximum length of a *symbolic-name* is 31 characters.

3.2.3 Constants.

R304	*constant*	**is** *literal-constant*
		or *symbolic-constant*
R305	*literal-constant*	**is** *int-constant*
		or *real-constant*
		or *complex-constant*
		or *logical-constant*
		or *char-constant*
		or *bit-constant*
R306	*symbolic-constant*	**is** *symbolic-name*

3.2.4 Operators.

R307	*intrinsic-operator*	**is** *power-op*
		or *mult-op*
		or *add-op*
		or *bnot-op*
		or *band-op*
		or *bor-op*
		or *concat-op*
		or *rel-op*
		or *not-op*
		or *and-op*
		or *or-op*
		or *equiv-op*
R308	*power-op*	**is** **
R309	*mult-op*	**is** *
		or /
R310	*add-op*	**is** +
		or −
R311	*bnot-op*	**is** .BNOT.
R312	*band-op*	**is** .BAND.
R313	*bor-op*	**is** .BOR.
		or .BXOR.
R314	*concat-op*	**is** //
R315	*rel-op*	**is** .EQ.
		or .NE.
		or .LT.
		or .LE.
		or .GT.
		or .GE.
		or ==
		or <>
		or <
		or <=
		or >

		or >=
R316	*not-op*	**is** .NOT.
R317	*and-op*	**is** .AND.
R318	*or-op*	**is** .OR.
R319	*equiv-op*	**is** .EQV. **or** .NEQV.
R320	*defined-operator*	**is** *overloaded-intrinsic-op* **or** *defined-unary-op* **or** *defined-binary-op*
R321	*overloaded-intrinsic-op*	**is** *intrinsic-operator*
R322	*defined-unary-op*	**is** . *letter* [*letter*]... .
R323	*defined-binary-op*	**is** . *letter* [*letter*]... .

Constraint: A *defined-unary-op* and a *defined-binary-op* must not contain more than 31 characters and must not be the same as any *intrinsic-operator* or *logical-constant*.

3.2.5 Statement Labels. Any statement may be labeled.

R324	*label*	**is** *digit* [*digit* [*digit* [*digit* [*digit*]]]]

In free source form (3.3.1), a label is considered a lexical element that must immediately precede the statement. In fixed source form (3.3.2), a label may appear only in character positions 1-5; blanks may appear within a fixed source form label. The same statement label must not be given to more than one statement in a program unit. Leading zeros are not significant in distinguishing between statement labels and blanks are not significant in distinguishing between statement labels in fixed source form.

3.2.6 Delimiters. The special characters blank, comma, equals, colon, left parenthesis, right parenthesis, left bracket, right bracket, percent, slash, and asterisk are used in various delimiting ways, as described in the syntax rules.

3.2.7 Lexical Element Sequence and Separation. The syntax rules specify the sequences of lexical elements that are valid Fortran statements. In general, any number of optional blanks may be inserted between otherwise adjacent lexical elements. In free source form (3.3.1), one or more blanks must separate a symbolic name adjacent to a keyword and blanks must not be inserted within any lexical element. In fixed source form (3.3.2), this is not the case; blanks may be inserted or omitted freely.

3.3 Source Form.

A Fortran program is a sequence of source records, called lines. These records contain the characters that make up the statements of a program unit. Lines following a program unit END statement are not part of that program unit.

Any syntax rule term that ends with "*-stmt*" is a Fortran statement.

A **character context** means characters within (between the delimiters for) character constants, format-item lists in FORMAT statements, and comments.

There are two **source forms**:, free and fixed. **Free form** has no character position restrictions and statements may appear in any character positions on the lines. Fixed form reserves character positions 1-6 of each source line for special purposes. Free form and fixed form must not be mixed in the same program unit. The means for specifying the source form of a program unit is processor dependent.

3.3.1 Free Source Form. In free form, each source record may contain from zero to a maximum of 132 characters. Blank characters are significant and must not appear within lexical elements and must be used to separate a symbolic name adjacent to a keyword. A sequence of blank characters outside of a character context is equivalent to a single blank character.

3.3.1.1 Commentary. The character "!" initiates a **comment** except when it appears within a character context. The comment extends to the end of the source line. A comment, including its "!" delimiter, is processed as though it were a blank character. Lines containing only blanks or blank equivalents are ignored and may appear anywhere in a program unit.

3.3.1.2 Statement Separation. The character ";" separates statements on a single source line except when it appears within a character context. A sequence of characters consisting of a ";" followed by blanks, blank equivalents, or another ";" (possibly with intervening blanks) is equivalent to a ";" except within a character context. A semicolon may follow the last statement on a line. A statement must not follow an END statement on the same line.

3.3.1.3 Statement Continuation. Outside of a comment, the character "&" as the last nonblank character on a line signifies that the statement is continued on the next line. If the first nonblank character on the next line is also "&", the statement continues at the next character position following the "&"; otherwise, it continues at character position 1. When used for continuation, the "&" is not part of the statement. If a character context other than a comment is being continued, the "&" signifying continuation cannot be followed by commentary and the continued portion must begin with an "&". If the continuation is not within a character context, the "&" may be followed by commentary. A statement must not contain more than 1320 characters.

3.3.2 Fixed Source Form. Fixed form is the same as free form, with the following exceptions:

(1) Blank characters outside of a character context are insignificant and may be used freely throughout the program.

(2) Source lines are exactly 72 character positions long.

(3) Lines with a "C" or "*" in character position 1 are additional forms of commentary.

(4) The "&" continuation is not used in fixed form; rather, character position 6 is used. If character position 6 contains a blank or zero, a new statement begins in character position 7 of this line and character positions 1-5 may contain a label. If character position 6 contains some character other than a blank or zero, character positions 7-72 of this line constitute a continuation of the preceding (noncomment) line. Columns 1-5 of such continuation lines must be blank. A statement must not have more than 19 continuation lines.

(5) Statement labels may appear only in character positions 1-5 and the continuation indicator may appear only in character position 6.

(6) The program unit END statement must not be continued and no other statement in the program unit may have an initial line that appears to be a program unit END statement.

4 DATA TYPES

A data entity is either a specific data value (e.g., the value of a specific expression) or an accessible entity that may contain a specific data value (e.g., a simple scalar variable). In either case, a data entity is associated with a specific instance of a data value. A data object is a data entity that has a name or is a constant. Data objects may also be collections of other data objects, as is the case with arrays and structured objects.

A data type does not represent specific instances of data values, but rather defines the properties of a specific class of data values and the allowed operations on them. For example, the data type integer defines the class of integer numeric values and the operations of integer arithmetic. Each data object has a data type. Data objects may have other attributes in addition to their types. How data types and other attributes are specified for data objects is described in Section 5.

There are two categories of data types: intrinsic types and derived types. An intrinsic type (e.g., integer) is one that is defined implicitly, along with operations, and is always available. A derived type is a data structure definition whose components are intrinsic types or other derived types. A derived type must be defined, whereas an intrinsic type is predefined. The distinction between data type and data object is especially important in the case of derived types and is reflected in the separate steps of type definition and object declaration.

4.1 The Concept of Type. A data type consists of a specific set of data values. The principal properties of such a class are: (1) the set of valid values and their representation (constants) and (2) the set of operations provided on and between these values.

4.1.1 Set of Values. For each data type, there is a set of valid values. The set of valid values may be completely specified, as is the case for bit or logical, or may be specified by a processor-dependent method, as is the case for integer and real. For data types such as complex or derived types, the set of valid values consists of the set of all the combinations of the values of the individual components.

4.1.2 Constants. For each of the intrinsic data types, the form for literal constants of that type is defined intrinsically. These literal constants are described below for each intrinsic type.

A constant value may be given a symbolic name.

Constants for derived types cannot be represented directly. Rather, a symbolic name may be given to a constant expression (7.1.6.1) formed from derived type values using constructors (4.4.2).

4.1.3 Operations. For each of the intrinsic data types, a set of operations and corresponding operators are defined intrinsically. These are described in Section 7. In addition, operations and operators may be defined, augmenting the intrinsic set. Operator definitions are described in Sections 7 and 12.

The only intrinsic operations for derived types are equality comparisons (.EQ. and .NE.). All other operations on derived type entities must be defined.

4.2 Assignment.

Assignment provides a means of defining or redefining the value of a variable of any type.

Assignment (7.5) is intrinsically defined for all types where there is conformance between the type and other attributes of the variable and the value to be assigned to it. Assignment is intrinsically defined with certain specific conversions, as described in Section 7, where the value and variables do not have to conform. For example, an integer value may be assigned to a real variable and the necessary conversion is applied. For nonintrinsic assignment, conversions may be defined by assignment subroutines (Section 7 and 12.5.2.3).

4.3 Intrinsic Data Types.

The intrinsic data types are:

numeric types:	Integer, Real, Complex, and Double Precision
nonnumeric types:	Character, Logical, and Bit

4.3.1 Numeric Types. The numeric types are provided for numerical computation. The normal operations of arithmetic, addition (+), subtraction (−), multiplication (*), division (/), exponentiation (**), negation (unary −), and identity (unary +), are defined intrinsically for all of these types.

Each numeric type includes a zero value, which is considered to be neither negative nor positive. In this standard, the unqualified term "constant" means "unsigned constant" when applied to numeric types.

4.3.1.1 Integer Type. The set of values for the integer type is a subset of the mathematical integers. This subset includes all of the integer values from some processor-dependent minimum negative value to some processor-dependent maximum positive value.

The type specifier (R502) for the integer type is the keyword INTEGER.

Any integer value may be represented as a *signed-int-constant*.

R401	*int-constant*	**is** *digit* [[_] *digit*]...
R402	*signed-int-constant*	**is** [*sign*] *int-constant*
R403	*sign*	**is** +
		or −

Examples of unsigned and signed integer constants are:

```
473
5_000_000
+56
-101
```

An underscore character in an integer constant is insignificant and has no effect on the value of the constant. An integer constant represents a decimal value.

4.3.1.2 Real and Double Precision Type. The real type approximates the mathematical real numbers. A processor must provide two or more **approximation methods** that define sets of values that are representable for data entities of type real. Each such method is characterized by an effective decimal precision and an effective decimal exponent range. The effective decimal precision of an approximation method is returned by the inquiry intrinsic function EFFECTIVE_PRECISION (13.9.38) and the effective decimal range is returned by the inquiry intrinsic function EFFECTIVE_EXPONENT_RANGE (13.9.37).

A data entity of type real may have **real type parameters** specified for precision and exponent range. The values specified for these type parameters indicate minimum requirements for the approximation method selected to represent the data object. A processor must select an approximation method with an effective decimal precision that is greater than or equal to the specified precision, and with an effective decimal exponent range that is greater than or equal to the specified exponent range. If more than one such method exists, the processor must select the method with effective decimal precision that exceeds the specified precision required by the least margin. If more than one method still exists, the processor must select the method with effective decimal exponent range that exceeds the specified exponent range by the least margin. If more than one method still exists, the method selected is processor dependent. If no method exists that satisfies the specified precision and exponent range, the processor must indicate an error condition, but other processor action is undefined.

If one of the type parameters is omitted in the specification of a data object of type real, a processor-dependent default is used.

If neither type parameter is specified, a processor-defined default real method is selected and the data object is of type **default real**.

If double precision is specified for a data object, a processor-defined double precision method is selected and the object is of type double precision. The effective decimal precision of the double precision method must be greater than that of the default real method.

A data object of specified precision or exponent range may select the default real or double precision method for its representation if either of these methods satisfies the specified requirements. Such a data object must not become associated with a default real or double precision data object in argument association (12.4.1.1) or in an IDENTIFY statement (6.2.6).

The type specifier for the real type is the keyword REAL and the type specifier for the double precision type is the keyword DOUBLE PRECISION.

R404	*signed-real-constant*	**is** [*sign*] *real-constant*
R405	*real-constant*	**is** *significand* [*exponent-letter exponent*]
		or *int-constant exponent-letter exponent*
R406	*significand*	**is** *int-constant* . [*int-constant*]
		or . *int-constant*
R407	*exponent*	**is** *signed-int-constant*
R408	*exponent-letter*	**is** E
		or D
		or *defined-exponent-letter*
R409	*exponent-letter-stmt*	**is** EXPONENT LETTER *precision-selector defined-exponent-letter*
R410	*defined-exponent-letter*	**is** *letter*

Constraint: A *defined-exponent-letter* must be a letter other than E, D, or H.

A given letter may be specified as the defined exponent letter in one and only one EXPONENT LETTER statement in a given declaration part sequence.

Real constants written without an exponent part, or with exponent letter E, are default real objects; exponent letter D specifies a double precision constant. A specified precision real constant must use the exponent character specified for that precision in an EXPONENT LETTER statement. A defined exponent letter and its association with a particular precision selector may be made accessible to a program unit by a USE statement.

Examples of signed real constants are:

```
-12.78
+1.6E3
2.1
```

Examples of unsigned real constants are:

```
0.45E-4
10.93L7
.123
3E4
```

In the second example (10.93L7), the letter L must have been defined as an exponent letter in an EXPONENT LETTER statement.

The exponent represents the power of ten scaling to be applied to the significand. The meaning of these constants is as in decimal scientific notation.

4.3.1.3 Complex Type. The **complex type** approximates the mathematical complex numbers. The values of a complex type are ordered pairs of real values. The first real value in a complex pair value is called the **real part**, and the second real value is called the **imaginary part**.

Any approximation method used to represent data objects of type real may be used to represent both the real and imaginary parts of a data object of type complex. The precision and exponent range type parameters may be specified for complex data objects. They express the required minimum precision and exponent range requirements for the real approximation method used to represent both the real and imaginary parts of the complex data object. The specified precision and exponent range select one real approximation method for both parts following the same rules as for the real type.

If neither the precision nor the exponent range is specified, the default real method is selected for both parts and the complex data object is **default complex**. A specified precision or exponent range complex data object must not be associated with a default complex object in argument association (12.4.1.1) or in an IDENTIFY statement (6.2.6).

The type specifier for the complex type is the keyword COMPLEX.

R411	*complex-constant*	**is** (*real-part* , *imag-part*)
R412	*real-part*	**is** *signed-int-constant*
		or *signed-real-constant*
R413	*imag-part*	**is** *signed-int-constant*
		or *signed-real-constant*

If the real part and imaginary part of a complex constant do not have the same precision and exponent range type parameters, both are converted to an approximation method consistent with the maximum of the two precisions and the maximum of the two exponent ranges.

If both the real and imaginary parts are signed integer constants, they are converted to the default real approximation method and the constant is of type default complex. If only one of the parts is a signed integer constant, the signed integer constant is converted to the approximation method selected for the signed real constant.

4.3.2 Nonnumeric Types. The nonnumeric types are provided for nonnumeric processing. The intrinsic operations defined for each of these types are indicated below.

4.3.2.1 Character Type. The **character type** is a set of values composed of character strings. A **character string** is a sequence of characters, numbered from left to right 1, 2, 3, ... up to the number of characters in the string. The number of characters in the string is called the **length** of the string. The length is a type parameter and its value must be greater than or equal to zero. Any character representable in the processor may occur in a character string. Strings of different lengths are all of type character.

The type specifier for the character type is the keyword CHARACTER.

Literal character constants are written as a sequence of characters, delimited by either apostrophes or quotation marks.

R414 *char-constant* **is** ' [*character*]... '
or " [*character*]... "

An apostrophe character within an character constant delimited by apostrophes is represented as two consecutive apostrophes (without intervening blanks); in this case, the two apostrophes are counted as one character. Similarly, a quotation mark character within a character constant delimited by quotation marks is represented as two consecutive quotation marks and the two quotation marks are counted as one character.

The intrinsic operation **concatenation** (//) is defined between two data objects of type character (7.2.3).

4.3.2.2 Logical Type. The **logical type** has two values which represent true and false.

R415 *logical-constant* **is** .TRUE.
or .FALSE.

The intrinsic operations defined for data objects of logical type are: negation (.NOT.), conjunction (.AND.), inclusive disjunction (.OR.), logical equivalence (.EQV.), and logical non-equivalence (.NEQV.). There is also a set of intrinsically defined relational operators that compare the value of data objects of other types and yield a logical value. These operations are described in Section 7.2.4.

The type specifier for the logical type is the keyword LOGICAL.

4.3.2.3 Bit Type. The **bit type** has two values which represent the bit values zero and one. Each of these values has two literal representations.

R416 *bit-constant* **is** B"0"
or B'0'
or B"1"
or B'1'

The intrinsic operations defined for data objects of bit type are: bit negation (.BNOT.), bit conjunction (.BAND.), bit inclusive disjunction (.BOR.), and bit exclusive disjunction (.BXOR.). These operations are described in 7.2.2.

The type specifier for the bit type is the keyword BIT.

4.4 Derived-Data Types.

Additional data types may be derived from the intrinsic data types. Each such derived type is defined as a set of components, where each component is an intrinsic type or another previously defined derived type. Ultimately, the structure of a derived type is resolved into a sequence of components of intrinsic type. Objects of derived type are called **structures** or **structured objects**.

4.4.1 Derived-Type Definition.

R417 *derived-type-def* **is** *derived-type-stmt*
component-def-stmt
[*component-def-stmt*]...
[*variant-component*]
end-type-stmt

R418 *derived-type-stmt* **is** [*access-spec*] TYPE *type-name* [(*type-param-name-list*)]

R419 *end-type-stmt* **is** END TYPE [*type-name*]

Constraint: A derived type *type-name* must not be the same as any intrinsic *type-name*.

Constraint: If END TYPE is followed by a *type-name*, the *type-name* must be the same as that in the *derived-type-stmt*.

R420 *component-def-stmt* **is** *type-spec* [[, *component-attr-spec*]... ::] *component-decl-list*

Constraint: A *type-spec* in a *component-def-stmt* must not contain a *type-param-value* that is an asterisk.

R421 *component-attr-spec* **is** PRIVATE
or ARRAY (*explicit-shape-spec-list*)

R422 *component-decl* **is** *component-name* [(*explicit-shape-spec-list*)]

R423 *variant-component* **is** *select-case-stmt*
[*case-stmt* [*component-def-stmt*]...]...
end-select-stmt

Constraint: The *select-case-stmt* and *end-select-stmt* in a *variant-component* must not specify a *construct-name*.

A derived-type name must not be the same as any intrinsic type name and must not be the same as any other accessible derived-type name. The type name is analogous to the intrinsic type names (e.g., INTEGER, CHARACTER) and specifies the derived type being defined.

An example of a derived-type definition is:

```
TYPE PERSON
   INTEGER AGE
   CHARACTER (LEN = 50) NAME
END TYPE PERSON
```

4.4.1.1 Type Parameters of Derived Type. Derived-type definitions may have type parameters that are symbolic names for integer values. These symbolic names may be used as parameters in the specification of expressions in the derived-type definition. In a declaration of a data object of a type whose definition contains type parameters, actual values for these parameters must be specified. This establishes the actual type parameter values for these objects.

Type parameters of derived type are analogous to precision and exponent range type parameters for the real and complex types and character length for the character type.

An example of a derived-type definition with type parameters is:

```
TYPE STRING (MAX_SIZE)
   INTEGER                    LENGTH
   CHARACTER (LEN = MAX_SIZE) VALUE
END TYPE STRING
```

4.4.1.2 Derived-Type Variant Component. A variant component specifies alternative sequences of components. Only one such sequence has an interpretation at any given time in a structured object of that type. The nonvariant component immediately preceding the variant component of a variant derived type is the **tag component**. The value of the tag component in a structured object determines which sequence of variant components is selected. The selection follows the rules for the CASE construct, except that nesting and construct names are prohibited (8.1.3).

An example of a variant structure is:

```
TYPE GEOMETRIC
   REAL                 X,Y
   REAL                 AREA
   CHARACTER (LEN = 10) SHAPE  ! TAG
   SELECT CASE    (SHAPE)      ! VARIANT COMPONENT
          CASE ('CIRCLE')   ;  REAL  RADIUS
          CASE ('SQUARE')   ;  REAL  SIDE
          CASE ('RECTANGLE');  REAL  HEIGHT, WIDTH
          CASE ('POLYGON')  ;  INTEGER NUM_EDGES;  REAL EDGES (10)
   END SELECT
END TYPE GEOMETRIC
```

4.4.1.3 Equivalence of Derived Types. A particular type name may be defined at most once in any program scope. Derived-type definitions with the same type name may appear in different program scopes, in which case they are independent and define different derived types.

Two data objects have the same type if they are declared with reference to the same derived-type definition; conversely, two objects are of different type if they reference different derived-type definitions, even if the two derived types have identical components defined in the same order.

4.4.2 Derived-Type Values. The set of values of a specific derived type consists of all combinations of the various component values. A value may be constructed from a corresponding sequence of values, one value for each component of the derived type. A derived-type definition also defines a corresponding derived-type constructor for that derived type.

R424 *derived-type-constructor* **is** *type-name* [(*type-param-spec-list*)] (*expr-list*)

Constraint: The *type-param-spec* option must be supplied if and only if the referenced type definition includes type parameters.

The sequence of expressions in a derived-type constructor specifies component values, which must agree in number, order, type, and shape with the components of the derived type. If necessary, each value is also coerced according to the rules of assignment so that its value has the same actual type parameters as those specified by *type-param-value*. A constructor whose values are all constant expressions is a derived-type constant expression. Using the derived types illustrated in 4.4.1.1 and 4.4.1.2, examples of derived-type constructor are:

```
STRING  (20) (19, 'NOW IS THE TIME FOR')
GEOMETRIC (0., 0., 4., 'SQUARE', 2.)
```

4.4.3 Operations on Derived Types. Any operations on derived-type data objects, other than the intrinsically defined equality comparisons (.EQ. and .NE.), must be defined by operator functions. Such definitions are made as described in Section 12. Function values and arguments (see Section 12) may be of any derived type.

Two objects of the same derived type with variant components may be compared, even if the value of their tag components are not equal; the result of a comparison with unequal tag components is that the objects are not equal.

4.5 Array Constructors. An **array constructor** is defined as a sequence of specified scalar values and interpreted as a rank-one array whose element values are those specified in the sequence. The sequence of values may be specified by any combination of individual scalar values, ranges of values, rank-one arrays, and other array constructors.

R425	*array-constructor*	**is** [*array-constructor-value-list*]
		or (/ *constructor-value-list* /)

In the preceding syntax rule, the brackets are part of the syntax.

R426	*array-constructor-value*	**is** *scalar-expr*
		or *rank-1-array-expr*
		or *scalar-int-expr* : *scalar-int-expr* [: *scalar-int-expr*]
		or [*int-constant-expr*] *array-constructor*

The *int-constant-expr* in the fourth form of *array-constructor-value* specifies the number of consecutive copies of the associated *array-constructor*. The type of an array constructor is the type of the scalar value interpreted as the first array element. Each subsequent scalar value in the sequence must have intrinsic assignment conformance as described in 7.5.1.4, and the value is so converted.

5 DATA OBJECT DECLARATIONS AND SPECIFICATIONS

Every data object has a type, a rank, and a shape and may also have a number of additional properties. These properties determine the characteristics of the data and the uses of the objects. Collectively these properties, including the type, are termed the **attributes** of the data object. A data object must not be explicitly specified to have a particular attribute more than once in a program unit. With the exception of literal constants which may be denoted directly in the program, every data object is denoted by a symbolic name. The type of a data object is either determined implicitly by the first letter of its name (5.3) or is specified explicitly in a **type declaration statement**. Some of the additional attributes may also be specified by separate specification statements; all of them may be included in a type declaration statement.

For example:

```
INTEGER  INCOME, EXPEND
```

declares the two data objects named INCOME and EXPEND to have the type integer.

```
REAL, ARRAY(-5:+5) :: X, Y, Z
```

declares three data objects with names X, Y, and Z. These all have default real type and are explicit-shape rank-one arrays with a lower bound of −5, an upper bound of +5, and a size of 11.

5.1 Type Declaration Statements.

R501	*type-declaration-stmt*	**is** *type-spec* [[, *attr-spec*]... ::] *object-decl-list*
R502	*type-spec*	**is** INTEGER
		or REAL [*precision-selector*]
		or DOUBLE PRECISION
		or COMPLEX [*precision-selector*]
		or CHARACTER [*length-selector*]
		or LOGICAL
		or BIT
		or TYPE (*type-name* [(*type-param-spec-list*)])
R503	*type-param-spec*	**is** [*type-param-name* =] *type-param-value*
R504	*type-param-value*	**is** *specification-expr*
		or *
R505	*attr-spec*	**is** *value-spec*
		or *access-spec*
		or ALIAS
		or ALLOCATABLE
		or ARRAY (*array-spec*)
		or INTENT (*intent-spec*)
		or OPTIONAL
		or RANGE
		or SAVE
R506	*object-decl*	**is** *object-name* [(*array-spec*)] [* *char-length*] [= *constant-expr*]

Constraint: No *attr-spec* may appear more than once in a given *type-declaration-stmt*.

Constraint: The *object-name* may be the name of an accessible data object or of a function procedure.

Constraint: The = *constant-expr* must appear if and only if the statement contains a *value-spec* attribute (5.1.2.1, 7.1.6.1).

Constraint: The * *char-length* option is permitted only if the *type-spec* is CHARACTER. If present **char-length* overrides the *length-selector* for that specific *object-decl* in which it appears.

Constraint: The ALLOCATABLE and RANGE attributes may be used only when declaring array objects.

Constraint: An array must not have both the ALLOCATABLE and the ALIAS attribute.

Constraint: An array specified with an ALIAS attribute must be declared with an *allocatable-spec*.

Constraint: The ALIAS attribute may be specified with type and array attributes only.

The value, accessibility, ALIAS, and SAVE attributes must not be specified for dummy arguments.

5.1.1 Type-Specifier Attributes. A **type specifier** specifies the type of all objects declared in an object declaration list. This type may override or confirm the implicit type indicated by the first letter of the object name as declared by the implicit typing rules in effect (5.3).

5.1.1.1 INTEGER. The INTEGER type specifier specifies that all objects whose names are declared in this statement are of intrinsic type integer (4.3.1.1).

5.1.1.2 REAL. The REAL type specifier specifies that all objects whose names are declared in this statement are of intrinsic type real (4.3.1.2). If a *precision-selector* is present, it has the form:

R507 *precision-selector* **is** (*type-param-value* [, [EXPONENT_RANGE =] □
□ *type-param-value*]
or (PRECISION = *type-param-value* □
□ [, EXPONENT_RANGE = *type-param-value*])
or (EXPONENT_RANGE = *type-param-value* □
□ [, PRECISION = *type-param-value*])

Constraint: The *type-param-value* must be an integer constant expression or an asterisk.

Let *p* be the value of the precision *type-param-value* and let *r* be the value of the exponent range *type-param-value*. Then the value of *p* is the minimum decimal precision and *r* is the minimum decimal exponent range required of the real approximation method used by the processor to implement the objects.

If either *p* or *r* is an asterisk, the objects being declared by the statement must be dummy arguments. The asterisk specifies that the corresponding *type-param-value* for the objects being declared is to be assumed from the actual arguments that become associated with the dummy arguments being declared.

If either part of the precision selector is omitted, a processor-dependent default value is used for the omitted type parameter.

If the precision selector is omitted, entirely, a processor-dependent default approximation method is selected and the objects declared are of the default real type.

5.1.1.3 DOUBLE PRECISION. The DOUBLE PRECISION type specifier specifies that objects whose names are declared in this statement are of intrinsic type double precision (4.3.1.2).

5.1.1.4 COMPLEX. The COMPLEX type specifier specifies that all objects whose names are declared in this statement are of intrinsic type complex (4.3.1.3).

The *precision-selector*, if present, is as for the real type (R507). The *precision-selector* specifies the minimum decimal precision and exponent range requirements for the real approximation method used by the processor to implement the two real values making up the real and imaginary parts of the complex value.

If the precision selector is omitted, the processor-dependent default real approximation method is used for both parts and objects declared are of default complex type.

5.1.1.5 CHARACTER. The CHARACTER type specifier specifies that all objects whose names are declared in this statement are of intrinsic type character (4.3.2.1). The length selector specifies the length of the character objects. The **char-length* may be part of an *object-decl*, in which case the length is specified for this single object and overrides the length specified in the length selector. If neither a length selector nor a **char-length* is specified, the length of the data object is 1.

R508	*length-selector*	**is**	[LEN =] *type-param-value*
		or	* *char-length [,]*
R509	*char-length*	is	(*type-param-value*)
		or	*scalar-int-constant*

If the type parameter value evaluates to a negative value, the length of character entities declared is zero.

A type parameter value of * may be used to declare a dummy argument of a procedure, in which case such a dummy argument assumes the length of the associated actual argument when the procedure is invoked.

A type parameter value of * may be used to declare symbolic constants, in which case the length is that of the constant values defined for the names.

An external function may be specified with a type parameter value of *; in this case any program unit invoking the function must declare this function name with a type parameter value other than *. When the function is invoked, the length of the result variable in the function is assumed from the value of this type parameter value.

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

5.1.1.6 LOGICAL. The LOGICAL type specifier specifies that all objects whose names are declared in this statement are of intrinsic type logical (4.3.2.2).

5.1.1.7 BIT. The BIT type specifier specifies that all objects whose names are declared in this statement are of intrinsic type bit (4.3.2.3).

5.1.1.8 Derived Type. A TYPE type specifier specifies that all objects whose names are specified in this statement are of the derived type specified by the type name in the *type-spec*. The declared objects have a component structure as defined by the *derived-type-def* (4.4.1).

Each type parameter value is associated with the corresponding type parameter name in a manner similar to the association of arguments in a procedure reference (12.4.1). The association may be positional or the type parameter names may be used as keywords, as with procedure arguments (Section 12).

A type parameter value of * may be used only with dummy arguments. The asterisk specifies that the relevant type parameter value is assumed from the associated object.

A declaration for a dummy argument object must specify a *derived-type-def* in a host procedure or module because the same definition must be used to declare both the actual and dummy arguments to ensure that both are of the same derived type.

5.1.2 Attributes. The additional attributes that may appear in the attribute specification of a type declaration statement further specify the nature of the objects being declared or specify restrictions on their use in the program.

5.1.2.1 Value Attribute. The *value-spec* specifies that the objects whose names are declared in the statement have a defined initial value. Those objects declared with a PARAMETER attribute are symbolic constants whose values must not be changed and those objects declared with the INITIAL attribute are variables whose values may be changed. The appearance of a *value-spec* in a specification requires that the =*constant-expr* option appear for all objects in the *object-decl-list*.

R510	*value-spec*	**is** PARAMETER
		or INITIAL

5.1.2.1.1 PARAMETER Attribute. The **PARAMETER attribute** specifies that objects whose names are declared in this statement are symbolic constants. The *object-name* becomes defined with the value determined from the *constant-expr* that appears on the right of the equals, in accordance with the rules of intrinsic assignment (7.5.1.4).

Any symbolic constant that appears in the constant expression must have been defined previously in the same type declaration statement, defined in a prior PARAMETER statement or type declaration statement using the PARAMETER attribute, or made accessible by an explicit or implicit USE statement.

A symbolic constant must not appear as part of a format specification.

5.1.2.1.2 INITIAL Attribute. The **INITIAL attribute** specifies that objects whose names are declared in this statement are variables whose values are initially defined. The *object-name* becomes defined with the value determined from the *constant-expr* that appears on the right of the equals, in accordance with the rules of assignment (7.5.1.4).

The presence of an INITIAL attribute implies that all the variables declared in this statement are saved. That is, INITIAL is equivalent to the combination INITIAL, SAVE.

5.1.2.2 Accessibility Attribute. The **accessibility attribute** specifies the accessibility of the objects in the *object-decl-list* to other external program units by a USE statement. The accessibility attribute may appear only in the *declaration-part* of a module.

R511	*access-spec*	**is** PUBLIC
		or PRIVATE

Objects that are declared with a PRIVATE attribute may be accessed only by procedures internal to the declaring program unit. Objects that are declared with a PUBLIC attribute may be made accessible in other external program units by the USE statement. The default for objects without an explicitly specified *access-spec* is PUBLIC, but this may be changed by a PRIVATE statement (see 5.2.3).

5.1.2.3 INTENT Attribute. The **INTENT attributes** may appear only within a procedure and may be specified only for dummy arguments. An intent attribute specifies that the objects whose names are declared in a statement including the attribute are dummy arguments to the procedure and specifies the intended use of the dummy argument within the procedure.

R512	*intent-spec*	**is** IN
		or OUT
		or INOUT

The INTENT (IN) attribute specifies that the dummy argument must not be redefined within the procedure.

The INTENT (OUT) attribute specifies that the dummy argument must be defined within the procedure before a reference to it is made and any actual argument that becomes associated with such a dummy argument must be definable. On invocation of the procedure, such a dummy argument becomes undefined.

The INTENT (INOUT) attribute specifies that the dummy arguments declared are intended for use both to receive data from and to return data to the invoking program unit. Any actual argument that becomes associated with such a dummy argument must be definable.

Objects declared with an INTENT attribute must not be also declared with a *value-spec*, *access-spec*, or SAVE attribute. Dummy procedures, dummy conditions, and allocatable arguments must not be declared with an INTENT attribute.

5.1.2.4 ARRAY Attribute. The **ARRAY attribute** specifies that objects whose names are declared in this statement are arrays with the same rank and shape specified by the *array-spec*. An *array-spec* may be part of an *object-decl*, in which case array properties are specified for this single object and override the *array-spec* in the ARRAY attribute. If the ARRAY attribute is omitted, an *array-spec* must be specified in the *object-decl* to declare an array object.

R513	*array-spec*	**is** *explicit-shape-spec-list*
		or *assumed-shape-spec-list*
		or *allocatable-spec-list*
		or *assumed-size-spec*

5.1.2.4.1 Explicit Shape Array. An **explicit shape array** is declared with an *explicit-shape-spec*. This specifies explicit values for the dimension bounds of the array.

R514	*explicit-shape-spec*	**is** [*lower-bound* :] *upper-bound*
R515	*lower-bound*	**is** *specification-expr*
R516	*upper-bound*	**is** *specification-expr*

Constraint: An explicit shape array whose bounds depend on the values of variables must either be a dummy argument or a local array of a procedure.

If any bound of a local array depends upon the value of a variable, such an array is termed **automatic**. An automatic array must not appear in a SAVE statement nor be declared with a SAVE attribute.

The values of the *specification-expr* determine the bounds of the array along a particular dimension and hence the extent of the array in that dimension. The declared subscript range of the array in that dimension is the set of integer values between and including the lower and upper bounds, provided the upper bound is not less than the lower bound. If the upper bound is less than the lower bound, the range is empty, the extent in that dimension is zero, and the array is of zero size. If the *lower-bound* is omitted, the value 1 is assumed.

The number of sets of bounds specified is the rank. The maximum rank is seven.

The declared bounds of an explicit shape array are the lower and upper bound. The declared shape is the shape determined by the declared bounds. The declared extents are the sizes determined by the declared bounds.

5.1.2.4.2 Assumed-Shape Array. An **assumed-shape array** is a dummy argument array that takes its shape from the associated actual argument array.

R517 *assumed-shape-spec* **is** [*lower-bound*] :

The size of a dimension of an assumed-shape array is the size of the corresponding dimension of the associated actual argument array. If the lower bound value is represented by d_1 and the size of the corresponding dimension of the associated actual argument array is s_a, then the value of the upper bound is $s_a + D_1 - 1$.

5.1.2.4.3 Allocatable Array. An **allocatable array** is one whose type, name, and rank are specified in a type declaration statement
containing an ALLOCATABLE attribute, but whose bounds, and hence shape, are declared when space is allocated for the array by execution of an ALLOCATE statement (6.2.2).

R518 *allocatable-spec* **is** :

The rank is equal to the number of colons in the *allocatable-spec-list*.

The size, bounds, and shape of an unallocated allocatable array are undefined, and no reference may be made to any part of it, nor may any part of it be defined. The declared lower and upper bounds of each dimension are those specified in the ALLOCATE statement when the array is allocatable.

An allocatable dummy array argument may be associated only with an allocatable actual argument. An actual argument which is an allocated allocatable array may be associated with a nonallocatable array dummy argument. A array-valued function may declare the result to be an allocatable array. A component of a derived type must not have the ALLOCATABLE attribute.

5.1.2.4.4 Assumed-Size Array. An **assumed-size** array is a dummy array where the size is assumed from that of an associated actual argument. The rank and extents may differ for the actual and dummy arrays; only the size of the actual array is assumed by the dummy array.

R519 *assumed-size-spec* is [*explicit-shape-spec-list* ,] [*lower-bound* :] *

Constraint: *assumed-size-spec* must not be included in an ARRAY attribute.

Constraint: The value to be returned by an array-valued function must not be declared as an assumed-size array.

The size of an assumed-size array is determined as follows:

(1) If the actual argument associated with the assumed-size dummy array is an array name of any type other than character, the size is that of the actual array.

(2) If the actual argument associated with the assumed-size dummy array is an array element of any type other than character with a subscript order value of r (6.2.4.2) in an array of size x, the size of the dummy array is MAX $(x - r + 1, 0)$.

(3) If the actual argument is a character array name, character array element name, or a character array element substring name (6.1.1), and if it begins at character storage unit t of an array with c character storage units, the size of the dummy array is MAX ((INT $(c - t + 1) / e$), 0), where e is the length of an element in the dummy character array.

If an assumed-size array has rank n, the product of the extents of the first $n - 1$ dimensions must be less than or equal to the size of the associated actual array.

An assumed-size array has no declared bounds, shape, or size.

5.1.2.5 SAVE Attribute. The **SAVE attribute** specifies that the objects declared in a declaration containing this attribute retain their definition status, effective range, and value after execution of a RETURN or END statement in the program unit containing the declaration. Such an object is called an **saved object**.

The SAVE attribute or SAVE statement may appear in declarations in a main program and has no effect.

Objects in a module program unit may be declared with a SAVE attribute. Such objects retain their definition status, effective range, and value when any procedure that accesses the module in a USE statement execute a RETURN or END statement. The SAVE attribute must not be specified for an object name that is in a common block.

5.1.2.6 OPTIONAL Attribute. The **OPTIONAL attribute** may be specified only for dummy arguments within a procedure subprogram. The OPTIONAL attribute specifies that such dummy arguments need not be associated with an actual argument in a reference to the procedure.

5.1.2.7 ALIAS Attribute. The **ALIAS attribute** specifies that only the type, rank, and name of the objects declared in the statement are specified. The object must not become definable until it is associated with a definable object as the result of executing an IDENTIFY statement (6.2.6).

5.1.2.8 RANGE Attribute. The **RANGE attribute** may be specified only for nonassumed-size array objects and specifies that those arrays may have their effective shapes changed by execution of SET RANGE statements. The initial effective shape of each array is its declared shape.

If the range list name is omitted, the arrays declared in that type declaration may have different shapes, and the individual array names may appear explicitly in SET RANGE statements. If the range list name is specified, the arrays must all be declared with the same rank, lower, bounds, and upper bounds and be reshaped only by execution of a SET RANGE statement containing that range list name.

5.2 Attribute Specification Statements.

Most of the attributes (other than type) may be specified for objects, independently of type, by single attribute specification statements. A data object must not be explicitly given any of the following attributes more than once in a program unit: type, value, accessibility, intent, array, save, optional, alias, and range.

5.2.1 INTENT Statement.

R520	*intent-stmt*	**is** INTENT (*intent-spec*) [::] *dummy-arg-name-list*

This statement specifies the intended use of the specified dummy arguments (5.1.2.3). Each specified dummy argument has the intent attribute.

5.2.2 OPTIONAL Statement.

R521	*optional-stmt*	**is** OPTIONAL [::] *dummy-arg-name-list*

This statement specifies that the specified dummy arguments need not be associated with actual arguments in a reference to the procedure (5.1.2.6). Each specified argument has the optional attribute.

5.2.3 Accessibility Statements.

R522	*access-stmt*	**is** *access-spec* [[::] *object-name-list*]

Constraint: An *access-stmt* may appear only in a module and only one accessibility statement with omitted object name list is permitted in a host program unit.

This statement declares the accessibility, PUBLIC or PRIVATE, of the object names (5.1.2.2). Each specified object name has the accessibility attribute.

If the object name list is omitted, the statement sets the default accessibility that applies to all potentially accessible objects in the module subprogram. For example, the statement

```
PUBLIC
```

confirms the standard default of public accessibility. The statement

```
PRIVATE
```

switches this default to objects being private unless individual objects are specified explicitly to be public.

5.2.4 SAVE Statement.

R523	*save-stmt*	**is** SAVE [[::] *saved-object-list*]
R524	*saved-object*	**is** *object-name*
		or / *common-block-name* /

Constraint: An object name must not be a dummy argument name, a procedure name, a function result name, an automatic array name, an alias name, or the name of an object in a common block.

Constraint: If a SAVE statement with an omitted saved object list occurs in a program unit, no other occurrence of the SAVE attribute or SAVE statement is permitted.

All objects named explicitly or included within a common block named explicitly have the SAVE attribute (5.1.2.5). If a particular common block name is specified in a SAVE statement in any subprogram of an executable program, it must be specified in a SAVE statement in every subprogram in which that common block appears. For a common block declared in a SAVE statement, the current values of the objects in a common block storage sequence (14.2.2) at the time a RETURN or END statement is executed are made available to the next program unit in the execution sequence of the executable program that specifies the common block name. 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 the named common block; a SAVE statement in the subprogram has no effect. The definition status of each object in the named common block storage sequence depends on the association that has been established for the common block storage sequence.

A SAVE statement with an empty saved object list is treated as though it contained the names of all objects in a program unit that may be saved.

5.2.5 DIMENSION Statement.

R525	*dimension-stmt*	**is** DIMENSION *array-name* (*array-spec*) [, *array-name* (*array-spec*)]...

Constraint: In a DIMENSION statement, only explicit shape and assumed-size *array-specs* are permitted.

This statement specifies a list of object names to have the ARRAY attribute and specifies the array properties that apply for each object named.

Each specified array name has the array attribute. The array properties for an array must not be specified in more than one of these statements in a program unit.

5.2.6 INITIALIZE Statement. The **INITIALIZE statement** provides a means of initially defining values for variables.

R526	*initialize-stmt*	**is** INITIALIZE (*initial-value-def-list*)
R527	*initial-value-def*	**is** *variable* = *constant-expr*

The variable becomes defined with the value determined from the constant expression that appears on the right of the equals in accordance with the rules of intrinsic assignment (7.5.1.4). A variable, or part of a *variable*, must not be initially defined more than once. A variable that appears in an INITIALIZE statement and is typed implicitly may appear in a subsequent declaration only if that subsequent type declaration confirms the implicit typing.

A variable that appears in an INITIALIZE statement has the SAVE attribute, but this may be reaffirmed by a SAVE statement or a type declaration statement containing the SAVE attribute.

5.2.7 PARAMETER Statement. The **PARAMETER statement** provides means of defining a symbolic constant with a constant value. Symbolic constants defined by a PARAMETER statement have exactly the same properties and restrictions as those declared in a type statement specifying a PARAMETER attribute (5.1.2.1.1).

R528	*parameter-stmt*	**is** PARAMETER (*symbolic-constant-def-list*)
R529	*symbolic-constant-def*	**is** *symbolic-constant-name* = *constant-expr*

The symbolic constant name must have its type, shape, and any type parameters specified either by previous occurrence in a type declaration statement in the same program unit, or must be determined by the implied typing rules currently in effect for the program unit. If the symbolic constant is typed by the implied type rules, its appearance in any subsequent type declaration statement must confirm this implied type and the values of any implied type parameters.

Each symbolic constant becomes defined with the value determined from the constant expression that appears on the right of the equals, in accordance with the rules of assignment (7.5.1.4).

A symbolic constant that appears in the constant expression must have been defined previously in the same PARAMETER statement, defined in a prior PARAMETER statement or type declaration statement using the PARAMETER attribute, or made accessible by an explicit or implicit USE statement.

Each symbolic constant has the parameter attribute.

5.2.8 CONDITION Statement. A **CONDITION statement** is used to specify a condition.

R530	*condition-stmt*	**is** CONDITION [::] *condition-name-list*

A condition name must not appear in any other statement in a declaration part sequence.

5.2.9 RANGE Statement. A **RANGE statement** specifies the RANGE attribute for each array name in the array name list.

R531	*range-stmt*	**is** RANGE [/ *range-list-name* /] *array-name-list*

If the range list name is present, the arrays in the array name list must all be declared with the same rank, lower bounds, and upper bounds. but they may be of any type. The effective shape of all arrays in the array name list may be changed by the execution of a SET RANGE statement containing only the range list name.

If the range list name is omitted, the arrays in the array name list may have different ranks, lower bounds, and upper bounds and each array name may appear in a SET RANGE statements. An array name must not be given the RANGE attribute more than once in a program unit.

Each array name appearing in a RANGE statement has the RANGE attribute.

5.3 IMPLICIT Statement. An IMPLICIT statement specifies a type, and possibly type parameters, for all implicitly typed data objects that begin with the letters specified in the statement. Alternatively, it may indicate that no implicit typing rules are to apply in a particular program unit.

R532	*implicit-stmt*	**is**	IMPLICIT *implicit-spec-list*
		or	IMPLICIT NONE
R533	*implicit-spec*	**is**	*type-spec* (*letter-spec-list*)
R534	*letter-spec*	**is**	*letter* [– *letter*]

A *letter-spec* consisting of two letters separated by a minus is equivalent to writing all of the letters in alphabetical order in the alphabetic sequence from the first letter through the second letter. For example, A–C is equivalent to A, B, C.

If IMPLICIT NONE is specified, all objects in the program unit must be explicitly declared.

Any data object not explicitly declared by a type declaration statement, or made accessible by a USE statement, that has a name starting with one of the letters in *letter-spec-list* is declared implicitly to be of type (and type parameters) of *type-spec.*

An IMPLICIT statement applies only to the program unit containing it. An IMPLICIT statement does not change the type of any intrinsic function. 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.

If no IMPLICIT statement is present, the default is equivalent to:

```
IMPLICIT INTEGER (I-N), REAL (A-H, O-Z)
```

However, if the external program unit contains a USE statement with the ONLY option omitted, no default implicit typing rules are defined. In such program units, all data objects must be explicitly declared or implicit typing rules must be established by an IMPLICIT statement.

5.4 Storage Association of Data Objects. In general, the physical storage units or storage order for data objects is not specifiable. However, the EQUIVALENCE statement and the COMMON statement provide for control of the "order" and "layout" of storage units. Section 14.2.2 describes the general mechanism of storage association.

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

If the equivalenced objects are of different data types, the EQUIVALENCE statement does not cause type conversion or imply mathematical equivalence. For example, if a scalar and an array are equivalenced, the scalar does not have array properties and the array does not have the properties of a scalar.

R535	*equivalence-stmt*	is	EQUIVALENCE *equivalence-set-list*
R536	*equivalence-set*	is	(*equivalence-object* , *equivalence-object-list*)
R537	*equivalence-object*	is	*object-name*
		or	*array-element*

or *substring*

Constraint: *object-name* must be a scalar variable name or an array variable name.

Constraint: An *equivalence-object* must not be the name of a dummy argument, an object of derived type, a component of an object of derived type, an allocatable array, an automatic array, an object of real type unless of default real type, an object of complex type unless of default complex type, an object of bit type, an array of zero size, or a function name.

Constraint: Each subscript or substring range expression in an *equivalence-object* must be an integer constant expression.

5.4.1.1 Equivalence Association. An EQUIVALENCE statement specifies that the storage sequences of the data objects whose names appear in an *equivalence-set* have the same first storage unit. This causes the association of the data objects in the *equivalence-set* and may cause association of other data objects.

5.4.1.2 Equivalence of Character Objects. A data object of type character may be equivalenced only with other objects of type character. The lengths of the equivalenced objects are not required to be the same.

An EQUIVALENCE statement specifies that the storage sequences of the character data objects whose names appear in an *equivalence-set* have the same first character storage unit. This causes the association of the data objects in the *equivalence-set* and may cause association of other data objects. Any adjacent characters in the associated data objects may also have the same character storage unit and thus may also be associated. In the example:

```
CHARACTER  (LEN=4) :: A, B
CHARACTER  (LEN=3) :: C(2)
EQUIVALENCE (A, C(1)), (B, C(2))
```

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

```
  1      2      3      4      5      6      7
 |---   --- A  ---    ---|
                     |---    --- B  ---    ---|
 |---   C(1)   ---|  |---    C(2)   ---|
```

5.4.1.3 Array Names and Array Element Names. If an array element name appears in an EQUIVALENCE statement, the number of subscripts must be the same as the rank of the array or one.

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

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

```
REAL ARRAY(2) :: A
REAL :: B
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))
```

5.4.2 COMMON Statement. The COMMON statement specifies blocks of physical storage, called **common blocks**, that may be accessed by any of the program units in an executable program. Thus, the COMMON statement provides a global data facility based on storage association (14.2.2). The common blocks specified by the COMMON statement may be named and are called **named common blocks** or may be unnamed and are called **blank common**.

R538 *common-stmt* is COMMON [/ [*common-block-name*] /]*common-block-object-list* □

□ [[,] / [*common-block-name*] /*common-block-object-list*]...

R539 *common-block-object* is *object-name* [(*explicit-shape-spec-list*)]

Constraint: *object-name* must be a *scalar-variable-name* or an *array-variable-name*. Only one appearance of object name is permitted in all common block object lists within a program unit.

Constraint: A *common-block-object* must not be the name of a dummy argument, an object of derived type, a component of an object of derived type, an allocatable array, an automatic array, an object of real type unless of default real type, an object of complex type unless of default complex type, an object of bit type, an array of zero size, or a function name.

Constraint: Each bound in the *explicit-shape-spec* must be an integer constant expression.

Each omitted common block name specifies the blank common block.

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

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

If a character variable or character array is in a common block, all of the entities in that common block must be of type character. Each array name appearing in a RANGE statement has the RANGE attribute.

5.4.2.1 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 data objects in the lists common block list for the common block. The order of the storage sequence is the same as the order of the appearance of the lists common block list 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. Data objects associated with an entity in a common block are considered to be in that common block.

5.4.2.2 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.

5.4.2.3 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 of entities in different program units.

5.4.2.4 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 causes data objects in named common blocks to become undefined unless the common block name has been declared in a SAVE statement, but never causes data objects in blank common to become undefined (14.3.2).

(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) A data object in a named common block may be initially defined by means of a DATA statement in a BLOCK DATA subprogram, but objects in blank common must not be initially defined (11.5).

5.4.2.5 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 object 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))
```

5.5 DATA Statement.

A DATA statement is used to provide initial values for variables.

R540	*data-stmt*	is	DATA *data-stmt-init* [[,] *data-stmt-init*]...
R541	*data-stmt-init*	is	*data-stmt-object-list* / *data-stmt-value-list* /
R542	*data-stmt-object*	is	*object-name*
		or	*array-element*
		or	*data-implied-do*
R543	*data-stmt-value*	is	[*data-stmt-repeat* *] *data-stmt-constant*
R544	*data-stmt-constant*	is	*constant*
		or	*signed-int-constant*
		or	*signed-real-constant*
R545	*data-stmt-repeat*	is	*int-constant*
		or	*scalar-int-symbolic-constant*
R546	*data-implied-do*	is	(*data-i-do-object-list*, *do-variable* = *scalar-int-expr*, *scalar-int-expr* [, *scalar-int-expr*])
R547	*data-i-do-object*	is	*array-element*
		or	*data-implied-do*

The data statement repeat factor must be a positive integer constant. If the data statement repeat factor is a symbolic constant, it must have been declared previously in the program unit or made accessible by a USE statement.

The variables or arrays whose names are included in the *data-i-do-list* must not be of type bit of a derived type, a dummy argument, made accessible by a USE statement, in a named common block unless the DATA statement is in a BLOCK DATA subprogram, in blank COMMON nor associated with an object in blank COMMON, a function name, or an alias object.

The arrays whose names are included in the *data-i-do-lists* must not be automatic arrays, allocatable arrays, or zero-sized arrays,

The *data-i-do-list* is expanded to form a sequence of scalar variables. Any array included is equivalent to a complete sequence of array elements, ordered by subscript order value (6.2.4.2). The *im-do-list* is repeated in the expanded *data-i-do-list* under control of the implied-do control index as in the DO loop (8.1.4.1, 9.4.2). The *array-element* must include a subscript (6.2.4) that depends on the value of *do-variable*. The *data-implied-do int-expr* may involve as primaries, constants, or inherited symbolic constants. The subscript expressions included in any array element names in the *data-i-do-list* must be expressions with primaries that are constants or the *do-var* of containing *implied-do-lists*.

The *data-stmt-value-list* is expanded to form a sequence of constant values. Each value must be either an explicit constant or a previously defined or inherited symbolic constant. A data statement repeat factor indicates the number of the following constant values are to be added to the sequence.

The expanded sequences of scalar variables and constant values are in one to one correspondence. Each constant defines the initial value for the corresponding variable. The lengths of the two expanded sequences must be the same.

The value of the constant must be assignment compatible with its corresponding variable, according to the rules of intrinsic assignment (7.5.1.2), and the constant defines the initial value of the variable according to those rules.

An initial value for a variable must be defined at most once in an executable program. If two or more variables are associated, only one may be initially defined in an executable program.

Examples of DATA statements are:

```
CHARACTER (LEN = 10)  NAME
INTEGER, ARRAY (0:9) :: MILES
REAL, ARRAY (100, 100) :: SKEW
DATA  NAME / 'JOHN DOE' /, MILES / 10*0 /
DATA  ((SKEW (I, J), I = 1, 100), J = I, 100) / 5050*0.0 /
DATA  ((SKEW (I, J), J = 1, 99), I = 1+J, 100) / 4950*1.0 /
```

The the character variable NAME is initialized with the value 'JOHN DOE', padding on the right since the length of the constant is smaller than the variable. All ten elements of the integer array MILES are initialized to zero, and the two dimensional array SKEW is initialized so that the lower triangle of SKEW is zero and upper triangle is one.

There must be the same number of items specified by each *data-stmt-object-list* and its corresponding *data-stmt-value-list*. There is a one-to-one correspondence between the items specified by a *data-i-do-list* and the constants specified by a *data-stmt-value-list* such that the first item of a *data-i-do-list* corresponds to the first constant of a list, etc. By this correspondence, the initial value is established and the data object 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 order value (6.2.4.2).

The type of the object item and the type of the corresponding constant must agree when either is of type character or logical. When the item is of type integer, real, double precision, or complex, the corresponding constant must also be of type integer, real, double precision, or complex; if necessary, the constant is converted to the type of the object according to the rules for numeric conversion and assignment (7.5.1.2). Note that if an object is of type double precision and the constant is of type real, the processor may supply more precision derived from the constant than can be contained in a real datum. A constant of type character is assigned to the object according to the rules for intrinsic assignment (7.5.1.2).

6 USE OF DATA OBJECTS

The appearance of a data object name in a context that requires its value is termed a **reference**. A reference is permitted only if the data object is defined (5.2.6, 5.2.7, 14.3.1). A data object becomes defined with a value when the data object name appears in certain contexts and when certain events occur (14.3).

A data object whose value may be redefined is a **variable**.

R601	*variable*	**is** *scalar-variable-name*
		or *array-variable-name*
		or *array-element*
		or *array-section*
		or *structure-component*
		or *substring*

Under some circumstances alias arrays (6.2.6), allocatable arrays (6.2.2), array sections (6.2.4.3), and variables associated with dummy arguments (7.5.1.1, 7.5.3.2, 12.4.1.1, 12.5.2.1, 12.5.2.7) must not be defined.

A literal constant is denoted by a syntactic form which indicates its type, shape, and value. A symbolic constant is a symbolic name that has been associated with a constant value with the PARAMETER attribute (5.1.2.1.1, 5.2.7). A reference to a constant is always permitted; redefinition of a constant is never permitted.

6.1 Scalars.

A **scalar** (2.4.4.1) is a data object whose value, if defined, is a single element from the set of values comprising its data type.

A scalar has rank zero.

6.1.1 Substrings.

A **substring** is a contiguous portion of a character string (4.3.2.1). The following rules define the forms of a substring:

R602	*substring*	**is** *parent-string* (*substring-range*)
R603	*parent-string*	**is** *char-scalar-variable-name*
		or *char-array-element*
		or *scalar-char-structure-component*
		or *scalar-char-symbolic-constant*
		or *scalar-char-constant*
R604	*substring-range*	**is** [*scalar-int-expr*] : [*scalar-int-expr*]

The first *scalar-int-expr* in *substring-range* is called the **starting point** and the second one is called the **ending point**. The length of a substring is the number of characters in the substring and is max (*ending-point* − *starting-point* + 1, 0).

Let the characters in the parent string be numbered $1, 2, 3, ..., n$, where n is the length of the parent string. Then the characters in the substring are those from the parent string from the starting point and proceeding in sequence up to and including the ending point. Both the starting point and the ending point must be within the range $1, ,2, ..., n$ unless the starting point exceeds the ending point, in which case the substring has length zero. The processor must signal the BOUND_ERROR condition if either the starting or ending point fails to meet these constraints and the substring occurs in an ENABLE construct (8.1.5) that specifies the BOUND_ERROR condition.

If the parent is a variable, the substring is also a variable. If the parent is an array section (6.2.4.3), the substring is an array of the same shape as the array section and each element is the designated substring of the corresponding element of the array section. A substring of an allocatable parent must not be allocated or deallocated. A substring of an aliased parent has the alias properties of the parent.

Examples:

`ID (4:9)`	scalar variable name as parent string
`'0123456789' (N:N)`	character constant as parent string

6.1.2 Structure Components. A derived-type definition contains one or more component definitions (4.4). A *structure-component* is one of the components of a structured object.

R605	*structure-component*	**is** *parent-structure* % *component-name* [*array-selector*]
R606	*parent-structure*	**is** *derived-type-scalar-variable-name*
		or *derived-type-array-variable-name*
		or *derived-type-array-element*
		or *derived-type-array-section*
		or *derived-type-structure-component*
		or *derived-type-symbolic-constant*

Constraint: An *array-selector* may appear only if the component specified by *component-name* is an array.

R607	*array-selector*	**is** (*subscript-list*)
		or (*section-subscript-list*)

The type of the structure component is the same as the type declared for the component in the derived-type definition.

The resulting data object has array properties if the parent or component has array properties.

If the parent has shape P and the selected component (including the array selector, if any) has shape C, the component will be an array of shape $[C, P]$, using the array constructor notation from Section 4.5. The remaining attributes are determined by the component declaration in the derived-type definition.

Examples:

`SCALAR_PARENT % SCALAR_FIELD`	scalar component of scalar parent
`ARRAY_PARENT (J) % SCALAR_FIELD`	component of array element parent
`ARRAY_PARENT (1:N) % SCALAR_FIELD`	component of array section parent
`SCALAR_PARENT % ARRAY_FIELD (K)`	array element component of scalar parent
`ARRAY_PARENT (K) % ARRAY_FIELD (J)`	array element component of array element parent
`ARRAY_PARENT % ARRAY_FIELD`	array component of array parent

6.2 Arrays. An **array** is a set of scalar data objects all having the same attributes (2.4.4.2). The scalar objects that make up an array are known as the **array elements**.

6.2.1 Whole Arrays. A **whole array** is an array name appearing without an appended parenthesized list and is referred to by its name.

6.2.1.1 Array Constants and Variables. A whole array is either a constant or variable. A **whole array constant** is the symbolic name of a constant expression (5.1.2.1.1 and 5.2.7) and comprises those elements determined by the declared shape of the symbolic constant.

The appearance of a whole array variable in an executable construct specifies those elements determined by the effective shape (6.2.1.2). A whole array variable that is an assumed-size array is permitted only as an actual argument in a procedure reference.

The appearance of a whole array name in a nonexecutable statement specifies the entire array determined by the declared shape.

No ordering of an elements of the array is indicated by the appearance of the array name, except when the name occurs in an input item (9.4.2), an output item (9.4.2), an initial value definition (5.2.6), an internal file unit (9.2.2), a format identifier (9.4.1.1) or a DATA statement object (5.5), where the order of reference is determined by the subscript order value (6.2.4.2).

6.2.1.2 Declared and Effective Array Range. The **declared range** for an array is the set of elements determined by the declared bounds for each dimension of the array. The **effective range** for an array is the subset of elements determined by the effective bounds of the array as specified in the most recently executed SET RANGE statement for the array. The **declared shape** for an array is the shape determined by the bounds of the array. The **effective shape** for an array is the shape determined by the effective range bounds of the array. If no SET RANGE statement has been executed for the array, the effective range is the declared range. The effective range of an array that is local to a program unit reverts to the declared range after execution of a RETURN or END statement in that program unit, unless the array has the SAVE attribute.

6.2.2 The ALLOCATE Statement. The **ALLOCATE statement** dynamically creates allocatable arrays.

R608	*allocate-stmt*	**is**	ALLOCATE (*array-allocation-list*)
R609	*array-allocation*	**is**	*array-name* (*explicit-shape-spec-list*)

Constraint: *array-name* must be the name of an allocatable array.

Constraint: The bound in an *array-allocation explicit-shape-spec* may be an arbitrary integer expression, but must not depend on any other bound in the same *allocate-stmt*.

Constraint: The number of *explicit-shape-spec*s in an *array-allocation explicit-shape-spec-list* must be the same as the declared rank of the array.

Example:

```
ALLOCATE (X (N), B (MAX (K, 0) : M, 0:9))
```

The values of the lower bound and upper bound expressions in an explicit shape specification determine the declared bounds of an allocatable array.

An allocatable array that has been allocated by an ALLOCATE statement and has not been subsequently deallocated (6.2.3) is **currently allocated** and is **definable**. Allocating a currently allocated array is prohibited. At the beginning of execution of an executable program, allocatable arrays have not been allocated and are **not definable**.

If the ALLOCATE statement appears in an ENABLE construct that specifies the ALLOCATION_ERROR condition, the processor must signal that condition if it is unable to allocate all of the arrays specified.

6.2.3 The DEALLOCATE Statement. The **DEALLOCATE statement** causes an allocatable array that has been allocated to become deallocated; hence, it becomes not definable.

R610 *deallocate-stmt* **is** DEALLOCATE (*array-name-list*)

The effect of deallocating an array that is not currently allocated is undefined. When the execution of a procedure is terminated by execution of a RETURN or END statement, all arrays allocated within the procedure except for allocatable dummy arguments, that are currently allocated and do not have the SAVE attribute are deallocated. Allocatable arrays that have the SAVE attribute retain their definition status upon execution of the RETURN or END statement.

An allocatable user-defined array-valued function is not deallocated upon execution of a RETURN or END statement. Such an allocatable array is deallocated by the processor after it has used the value returned by the function.

6.2.4 Subsets of Arrays. A **subset of an array** is either an array element, which is a scalar, or an array section, which is an array.

R611 *array-element* **is** *parent-array* (*subscript-list*)

Constraint: The number of subscripts must equal the declared rank of the array.

R612 *array-section* **is** *parent-array* (*section-subscript-list*) [(*substring-range*)]

R613 *parent-array* **is** *array-variable-name*
or *array-symbolic-constant-name*

Constraint: At least one *section-subscript* must be a *subscript-triplet* or a *vector-int-expr*.

Constraint: The number of *section-subscripts* must equal the declared rank of the array.

R614 *subscript* **is** *scalar-int-expr*

R615 *section-subscript* **is** *subscript*
or *subscript-triplet*
or *vector-int-expr*

Constraint: A *vector-int-expr section-subscript* must be a rank one integer array.

R616 *subscript-triplet* **is** [*subscript*] : [*subscript*] [: *stride*]

R617 *stride* **is** *scalar-int-expr*

6.2.4.1 Array Elements. The values of a subscript expression in an array element must be within the declared subscript range for that dimension.

If any subscript is outside the corresponding bounds of an array and the array element occurs in an ENABLE construct (8.1.5) that specifies the BOUND_ERROR condition, the processor must signal the BOUND_ERROR condition.

6.2.4.2 Subscript Order Value. The elements of an array form a sequence known as the **array element ordering**. The position of an array element in this sequence is determined by the subscript order value of the subscript list designating the element. The subscript order value is computed from the formulas in Table 6.1.

Table 6.1. Subscript Order Value

Rank *n*	Explicit Shape Specifier	Subscript List	Subscript Order Value

1	$j_1{:}k_1$	s_1	$1+(s_1-j_1)$
2	$j_1{:}k_1,j_2{:}k_2$	s_1,s_2	$1+(s_1-j_1)$ $+(s_2-j_2)\times d_1$
3	$j_1{:}k_1,j_2{:}k_2,j_3{:}k_3$	s_1,s_2,s_3	$1+(s_1-j_1)$ $+(s_2-j_2)\times d_1$ $+(s_3-j_3)\times d_2\times d_1$
· · ·	· · ·	· · ·	· · ·
n	$j_1{:}k_1,\ldots,j_n{:}k_n$	$s_1,\ldots,s_n$	$1+(s_1-j_1)$ $+(s_2-j_2)\times d_1$ $+(s_3-j_3)\times d_2\times d_1$ $+\cdots$ $+(s_n-j_n)\times d_{n-1}$ $\times d_{n-2}\times\cdots\times d_1$

Notes for Table 6.1:

(1) $d_i = \max\,(k_i - j_i + 1,\, 0)$ is the size of the ith dimension.

(2) $j_i \le s_i \le k_i$ for all $i = 1, 2, \ldots, n$.

6.2.4.3 Array Sections. An **array section** is an array data object designated by an array name with a section subscript.

Each nonscalar item in the section subscript list indicates a sequence of subscripts (6.2.4.4, 6.2.4.5). The array section is the set of elements from the named array determined by all possible subscript lists obtainable from the single subscripts or sequences of subscripts specified by each section subscript.

The rank of the array section is the number of nonscalar items in the section subscript list. The shape is the rank one array whose ith element is the number of integer values in the sequence indicated by the ith nonscalar item in the section subscript list. If any of these sequences is empty, the array section has size zero. The subscript order of the elements of an array section is that of the array data object that the array section represents.

If any subscript value in a section subscript is outside the corresponding declared range of the named array and the reference appears in an ENABLE construct (8.1.5) that specifies the BOUND_ERROR condition, the processor must signal the BOUND_ERROR condition.

6.2.4.4 Triplet Notation. The subscripts and strides of subscript triplet are optional. An omitted first subscript in a subscript triplet is equivalent to a subscript whose value is the effective lower bound for the named array and an omitted second subscript is equivalent to the effective upper bound (5.1.2.4, 5.1.4.2, 6.2.6). An omitted stride is equivalent to a stride of one.

The second subscript must not be omitted in the last dimension of an assumed-size array.

When the stride is positive, the subscripts specified by a triplet form a regularly spaced sequence of integers beginning with the first subscript and proceeding in increments of the stride to the largest such integer not exceeding the second subscript; the sequence is empty if the first subscript exceeds the second.

The stride must not be zero.

When the stride is negative, the sequence begins with the first subscript and proceeds in increments of the stride down to the smallest such integer equal to or exceeding the second subscript; the sequence is empty if the second subscript exceeds the first.

For example, if an array is declared as B (10), the array section B (3 : 11, 7) is the array of shape [2] consisting of the elements B (3) and B (10), in that order. The section B (9 : 1 : −2) is the array of shape [5] whose elements are B (9), B (7), B (5), B (3), and B (1), in that order.

For another example, suppose an array is declared as A (5, 4, 3). The section A (3 : 5, 2, 1 : 2) is the array of shape [3, 2] shown below:

```
A (3, 2, 1)   A (3, 2, 2)
A (4, 2, 1)   A (4, 2, 2)
A (5, 2, 1)   A (5, 2, 2)
```

6.2.4.5 Vector Subscripts. A section subscript that is a rank-one integer expression designates a sequence of subscripts that are the values of the expression; each element of the expression must be defined. The sequence is empty if the expression is of size zero.

For example, suppose Z is a two-dimensional array of shape [5, 7] and U and V are one-dimensional arrays of shape [3] and [4], 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 from 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 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)
```

Because Z (U, V) contains duplicate elements from Z, the section Z (U, V) must not be redefined.

6.2.5 The SET RANGE Statement. Execution of a **SET RANGE statement** establishes the effective range for the arrays in the array name list or for the members of the range list specified by the range list name.

R618	*set-range-stmt*	**is** SET RANGE ([*effective-range-list*]) *array-name-list*
		or SET RANGE ([*effective-range-list*]) / *range-list-name* /
R619	*effective-range*	**is** *explicit-shape-spec*
		or [*lower-bound*] : [*upper-bound*]

Constraint: The number of effective ranges in an *effective-range-list* must equal the rank of the arrays being ranged.

Constraint: All arrays being ranged must have the same lower bounds.

Constraint: An array that is a member of a range list must not appear in an *array-name-list* of a SET RANGE statement.

Each effective range specifies the effective lower and upper bounds for each array in *array-name-list* or *range-list*.

An array name must not appear in the array name list of a SET RANGE statement unless it has the RANGE attribute. A SET RANGE statement must not be used to establish the effective range for an allocatable or alias array that is not definable. The values of the effective lower bound and the effective upper bound must be within the declared bounds for the corresponding dimension of every array in the array list or every member of the range list specified by the range list name. The effect of a SET RANGE is global to all program units accessing those arrays by a USE statement. If the effective lower bound or the effective upper bound is omitted, they default to the current effective lower bound or effective upper bound, respectively. If the effective range list is omitted, the effective range is set to the declared range for the arrays in the array list or for the members of the range list specified by the range list name.

6.2.6 The IDENTIFY Statement. An **IDENTIFY statement** provides a dynamic aliasing facility involving an alias object and a parent object. An alias may be an array whose elements are a subset of the elements of a given parent. Such an alias has properties similar to those of an array section, but can specify a greater variety of subsets of the array elements of the parent. For example, an alias may be the diagonal of an array of rank two, or may have one subscript selecting an array of derived type and another indexing a component of the array elements (Examples 2 and 3 below).

R620	*identify-stmt*	**is** IDENTIFY (*alias-name* = *parent*)
		or IDENTIFY (*alias-element* = *parent-element* , □
		□ *alias-range-spec-list*)

Constraint: The alias and parent objects must conform in type, rank, and type parameters.

Constraint: The alias object must have the alias attribute and its name must not be the same as the *parent-name*.

R621	*alias-element*	**is** *alias-name* (*subscript-range*)

Constraint: The number of *subscript-name*s in an alias element must equal the number of *alias-range-spec*s.

Constraint: A subscript name must be a scalar integer variable name.

R622	*parent-element*	**is** *parent-name* (*subscript-mapping*) [% *component-name* [(*subscript-list*)
R623	*subscript-mapping*	**is** *subscript-list*

Constraint: Each *subscript* must be linear in the *alias-element subscript-name*s.

R624	*alias-range-spec*	**is** *subscript-range* = *subscript* : *subscript*

Constraint: The subscript ranges in a *subscript-name-list* must be identical to the subscript ranges in the corresponding alias range specification list, and must appear in the same order. A name must not appear more than once in such a list.

Constraint: Each integer expression in the subscript mapping must be linear in the subscript names.

An alias is **definable** following a valid execution of an IDENTIFY statement. An alias must not be defined unless it is definable. Execution of an IDENTIFY statement for an alias array that has the RANGE attribute sets the actual range and the effective range of the alias array to bounds specified by the range in the IDENTIFY statement.

The scope of the subscript names is the IDENTIFY statement itself, and the subscripts are implicitly of type integer.

The elements of the alias are specified by the subscript names varying over the corresponding ranges. The IDENTIFY statement specifies the mapping between the elements of the alias and the elements of the parent.

The linear mappings in the subscript lists of the parent elements must be mathematically equivalent to expressions of the form $k_0 + k_1 \times i_1 + k_2 \times i_2 + \cdots + k_n \times i_n$ where each k_j is a scalar integer expression not involving i_j and each i_j is named in the subscript name list. The mapping is established by evaluating $k_0, k_1, ..., k_n$.

If the parent is not an alias, the new alias is regarded as belonging to its parent. If the parent is an alias, it must be definable and the new alias is regarded as belonging to the nonalias object to which the parent belongs. If the parent is an allocatable array, it must be definable. Whenever an allocatable array is deallocated, all aliases belonging to it become not definable. On return from a procedure, all aliases established by an IDENTIFY statement within that procedure become not definable.

An alias array is said to be **many-to-one** if two or more of its elements are mapped onto the same parent element to which the alias belongs. If an alias is definable, it may be used according to the rules that govern the use of data objects, except that if it is many-to-one, the elements sharing a common parent element must not be defined or redefined.

When an alias array or a section of an alias array is associated with a dummy argument of a procedure, only elements within the alias array or alias array section are associated with the dummy argument.

The following are examples of aliasing:

(1) Simple alias

```
IDENTIFY (PART = STRUCTURE % COMPONENT)
```

(2) Skew section

```
IDENTIFY (DIAG (I) = ARRAY (I, I), I = 1:N)
```

(3) Array of structure components

```
IDENTIFY (PART (I) = STRUCTURE % ARRAY (I), I = 1:N)
IDENTIFY (PATTERN (I, J) = STRUCTURE (I) % ARRAY (J), I = 1:M, J = 1:N)
```

6.2.7 Summary of Array Name Appearances.

Table 6.2. Allowed Appearances of Array Names

Place of Appearance	Simple Array	Alias Array	Component Array	Allocatable Array
dummy-arg	Yes	No	No	Yes
use-stmt	Yes	Yes	No	Yes
type-declaration	Yes	Yes	No	Yes
equivalence-stmt	Yes	No	No	No
data-stmt	Yes	No	No	No

common-stmt	Yes	No	No	No
actual-arg in a reference to a procedure-subprogram	Yes	Yes	Yes	Yes
io-list	Yes	Yes	Yes	Yes
internal-file-id	Yes	Yes	Yes	Yes
fmt-spec	Yes	Yes	Yes	Yes
save-stmt	Yes	No	No	Yes
primary	Yes	Yes	Yes	Yes
assignment-stmt	Yes	Yes	Yes	Yes
identify-stmt	Yes	Yes	Yes	Yes
allocate-stmt	Yes	Yes	No	Yes
free-stmt	Yes	Yes	No	Yes

7 EXPRESSIONS AND ASSIGNMENT

This section describes the formation, interpretation, and evaluation rules for expressions and the assignment statement.

7.1 Expressions. An **expression** represents a computation, the result of which is either a scalar or an array object. An expression is formed from operands, operators, and parentheses. Simple forms of an operand are constants and variables, such as:

```
3.0
.FALSE.
A
B(I)
C(I:J)
```

An operand is either a scalar or an array. An operation is either intrinsic (7.2) or defined (7.3). More complicated expressions can be formed using operands which are themselves expressions.

7.1.1 Form of an Expression. Evaluation of an expression produces a value, which has a type, type parameters (if appropriate), and a shape (7.1.4).

Examples of expressions are:

```
A+B
(A-B)*C
A**B
C.AND.D
F//G
```

An expression is defined in terms of several categories: primary, level 1 expression, level 2 expression, level 3 expression, level 4 expression, level 5 expression, and level 6 expression.

These categories are related to the different operator precedence levels and, in general, defined in terms of other categories. The simplest form of each expression category is a *primary*. The rules given below specify the syntax of an expression. The semantics are specified in 7.2 and 7.3.

7.1.1.1 Primary.

R701	*primary*	**is** *constant*
		or *variable*
		or *array-constructor*
		or *derived-type-constructor*
		or *function-reference*
		or (*expr*)

Examples of a *primary* are:

Example	Syntactic Class
`1.0`	*constant*
`A`	*variable*
`[1.0,2.0]`	*array-constructor*
`PERSON('Jones', 12)`	*derived-type-constructor*
`F(X,Y)`	*function-reference*

(S+T)	*(expr)*

7.1.1.2 Level-1 Expressions. Defined unary operators have the highest operator precedence (Table 7.1). Level-1 expressions are primaries optionally operated on by defined unary operators:

R702	*level-1-expr*	**is** [*defined-unary-op*] *primary*
R322	*defined-unary-op*	**is** . *letter* [*letter*]... .

Simple examples of a *level-1-expr* are:

Example	Syntactic Class
A	*primary*
.INVERSE. B	*level-1-expr*

A more complicated example of a level-1 expression is:

```
.INVERSE. (A + B)
```

7.1.1.3 Level-2 Expressions. Level-2 expressions level-1 are expressions optionally involving the numeric operators *power-op*, *mult-op*, and *add-op*.

R703	*mult-operand*	**is** *level-1-expr* [*power-op mult-operand*]
R704	*add-operand*	**is** [*add-operand mult-op*] *mult-operand*
R705	*level-2-expr*	**is** [*add-op*] *add-operand*
		or *level-2-expr add-op add-operand*
R308	*power-op*	**is** **
R309	*mult-op*	**is** *
		or /
R310	*add-op*	**is** +
		or −

Simple examples of a level-2 expression are:

Example	Syntactic Class
A	*level-1-expr*
B ** C	*mult-operand*
D * E	*add-operand*
F − I	*level-2-expr*
+1	*level-2-expr*

A more complicated example of a level-2 expression is:

```
− A + D * E + B ** C
```

7.1.1.4 Level-3 Expressions. Level-3 expressions are level-2 expressions optionally involving the bit operators *bnot-op*, *band-op*, and *bor-op*.

R706	*band-operand*	**is** [*bnot-op*] *level-2-expr*
R707	*bor-operand*	**is** [*bor-operand band-op*] *band-operand*
R708	*level-3-expr*	**is** [*level-3-expr bor-op*] *bor-operand*
R311	*bnot-op*	**is** .BNOT.
R312	*band-op*	**is** .BAND.

R313	*bor-op*	**is** .BOR.
		or .BXOR.

Simple examples of a level-3 expression are:

Example	Syntactic Class
`A`	*level-2-expr*
`.BNOT. B`	*band-operand*
`C .BAND. D`	*bor-operand*
`E .BOR. F`	*level-3-expr*
`G .BXOR. H`	*level-3-expr*

A more complicated example of a level-3 expression is:

```
A .BXOR. B .BAND. .BNOT. C
```

7.1.1.5 Level-4 Expressions. Level-4 expressions are level-3 expressions optionally involving the character operator *concat-op*.

R709	*level-4-expr*	**is** [*level-4-expr concat-op*] *level-3-expr*
R314	*concat-op*	**is** //

Simple examples of a level-4 expression are:

Example	Syntactic Class
`A`	*level-3-expr*
`B // C`	*level-4-expr*

A more complicated example of a level-4 expression is:

```
X // Y // 'ABCD'
```

7.1.1.6 Level-5 Expressions. Level-5 expressions are level-4 expressions optionally involving the relational operators *rel-op*.

R710	*level-5-expr*	**is** [*level-4-expr rel-op*] *level-4-expr*
R315	*rel-op*	**is** .EQ.
		or .NE.
		or .LT.
		or .LE.
		or .GT.
		or .GE.
		or ==
		or <>
		or <
		or <=
		or >
		or >=

Simple examples of a level-5 expression are:

Example	Syntactic Class
`A`	*level-4-expr*
`B .EQ. C`	*level-5-expr*
`D < E`	*level-5-expr*

A more complicated example of a level-5 expression is:

```
(A + B) .NE. C
```

7.1.1.7 Level-6 Expressions. Level-6 expressions are level-5 expressions optionally involving the logical operators *not-op*, *and-op*, *or-op*, and *equiv-op*.

R711	*and-operand*	**is** [*not-op*] *level-5-expr*
R712	*or-operand*	**is** [*or-operand and-op*] *and-operand*
R713	*equiv-operand*	**is** [*equiv-operand or-op*] *or-operand*
R714	*level-6-expr*	**is** [*level-6-expr equiv-op*] *equiv-operand*
R316	*not-op*	**is** .NOT.
R317	*and-op*	**is** .AND.
R318	*or-op*	**is** .OR.
R319	*equiv-op*	**is** .EQV.
		or .NEQV.

Simple examples of a level-6 expression are:

Example	Syntactic Class
`A`	*level-5-expr*
`.NOT. B`	*and-operand*
`C .AND. D`	*or-operand*
`E .OR. F`	*equiv-operand*
`G .EQV. H`	*level-6-expr*
`S .NEQV. T`	*level-6-expr*

A more complicated example of a level-6 expression is:

```
A .AND. B .EQV. .NOT. C
```

7.1.1.8 General Form of an Expression. The general form of an expression involves the defined binary operators, which have the lowest precedence, and includes the previous categories of expressions.

R715	*expr*	**is** [*expr defined-binary-op*] *level-6-expr*
R323	*defined-binary-op*	**is** . *letter* [*letter*]... .

Simple examples of an expression are:

Example	Syntactic Class
`A`	*level-6-expr*
`B .UNION. C`	*expr*

More complicated examples of an expression are:

```
(B .INTERSECT. C) .UNION. (X-Y)
A+B .EQ. C*D
.INVERSE. (A + B)
A + B .AND. C * D
E // G .EQ. H(1:10)
```

7.1.2 Intrinsic Operations. An **intrinsic operation** is either an intrinsic unary operation or an intrinsic binary operation. An **intrinsic unary operation** is an operation of the form *intrinsic-operator* x_2 where x_2 is of an intrinsic type (4.3) which matches the type of the operand for the *intrinsic-operator* given in Table 7.1.

An **intrinsic binary operation** is an operation of the form x_1 *intrinsic-operator* x_2 where either x_1 and x_2 are of the intrinsic types (4.3) which match the types of the operands for the *intrinsic-operator* given in Table 7.1, and are in shape conformance (7.1.5), or x_1 and x_2 are of the same derived-type (4.4), are in shape conformance (7.1.5), and the *intrinsic-operator* is one of the relational operators .EQ., .NE., = =, or < >.

An **intrinsic operator** is the operator in an intrinsic operation.

A **numeric intrinsic operation** is an intrinsic operation for which the *intrinsic-operator* is a *numeric-operator* (+, −, *, /, or **). A **numeric intrinsic operator** is the operator in a numeric intrinsic operation.

For numeric intrinsic binary operations, the two operands may be of different numeric types or different type parameters. Except for a value raised to an integer power, if the operands do not have the same types or type parameters, each operand that differs in type or type parameters from those of the result is converted to the type and type parameters of the result before the operation is performed. When a value of type real, double precision, or complex is raised to an integer power, the integer operand need not be converted.

A **bit intrinsic operation, character intrinsic operation, relational intrinsic operation**, and **logical intrinsic operation** are similarly defined in terms of a *bit intrinsic operator* (.BAND., .BOR., .BXOR., and .BNOT.), *character intrinsic operator* (//), *relational intrinsic operator* (.EQ., .NE., .GT., .GE., .LT., .LE., = =, < >, >, > =, <, and < =), and *logical intrinsic operator* (.AND., .OR., .NOT., .EQV., and .NEQV.), respectively.

A **numeric relational intrinsic operation** is a relational intrinsic operation where the operands are of numeric type. A **character relational intrinsic operation** is a relational intrinsic operation where the operands are of type character. A **bit relational intrinsic operation** is a relational intrinsic operation where the operands are of type bit and the operator is .EQ., .NE., = =, or < >. A **derived-type relational intrinsic operation** is a relational intrinsic operation where the operands are of the same derived type and the operator is .EQ., .NE., = =, or < >.

Table 7.1. Type of Operands and Result for the Intrinsic Operation [x_1] *op* x_2. (The symbols I, R, D, Z, B, C, L, and Dt stand for the types integer, real, double precision, complex, bit, character, logical, and derived-type, respectively. Where more than one type for x_2 is given, the type of the result of the operation is given in the same relative position in the next column.)

Intrinsic Operator *op*	Type of x_1	Type of x_2	Type of [x_1] *op* x_2
unary +, −		I, R, D, Z	I, R, D, Z
binary +, −, *, /, **	I	I, R, D, Z	I, R, D, Z
	R	I, R, D, Z	R, R, D, Z
	D	I, R, D, Z	D, D, D, Z
	Z	I, R, D, Z	Z, Z, Z, Z
.BNOT.		B	B
.BAND., .BOR., .BXOR.	B	B	B
//	C	C	C

.EQ., .NE., = =, < >	I	I, R, D, Z	L, L, L, L
	R	I, R, D, Z	L, L, L, L
	Z	I, R, D, Z	L, L, L, L
	D	I, R, D, Z	L, L, L, L
	C	C	L
	B	B	L
	Dt	Same as x_1	L
.GT., .GE., .LT., .LE. >, > =, <, < =	I	I, R, D	L, L, L
	R	I, R, D	L, L, L
	D	I, R, D	L, L, L
	C	C	L
.NOT.		L	L
.AND., .OR., .EQV., NEQV.	L	L	L

7.1.3 Defined Operations. A **defined operation** is either a defined unary operation or a defined binary operation. A **defined unary operation** is an operation either of the form *defined-unary-op* x_2 where there exists a function subprogram accessible to the program unit containing *defined-unary-op* x_2 that specifies the operation (7.3) for the operator *defined-unary-op*, or of the form *intrinsic-operator* x_2 where the type of x_2 does not match that for the *intrinsic-operator* given in Table 7.1, and there exists a function subprogram accessible to the program unit containing *intrinsic-operator* x_2 that specifies the operation (7.3) for the operator *intrinsic-operator*.

A **defined binary operation** is an operation either of the form x_1 *intrinsic-operator* x_2 where there exists a function subprogram accessible to the program unit containing x_1 *intrinsic-operator* x_2 that specifies the operation (7.3), or of the form x_1 *intrinsic-operator* x_2 where the types and/or shapes of x_1 and x_2 are not those required for a binary intrinsic operation (7.1.2), and there exists a function subprogram accessible to the program unit containing x_1 *intrinsic-operator* x_2 that specifies the operation (7.3).

Note that an intrinsic operator can be defined as a nonintrinsic operation. In such a case, the intrinsic operator is said to be an **overloaded intrinsic operator.**

A **defined operator** is the operator in a defined operation.

An **extension operation** is a defined operation in which the operator is of the form *defined-op* (unary or binary). Note that the operator used in an extension operation may be overloaded in that more than one function subprogram accessible to the program unit specifying the same extension operation *defined-operator* may exist.

7.1.4 Data Type, Type Parameters, and Shape of an Expression. The data type and shape of an expression depend on the operators and on the data types and shapes of the primaries used in the expression, and are determined recursively from the syntactic form of the expression. The data type of an expression must be one of the intrinsic types (4.3) or of a derived type (4.4).

Only an expression whose type is real, double precision, complex, or character has type parameters. The type parameters are determined recursively from the form of the expression. The type parameters for an expression of type real, double precision, or complex are its precision and range parameters. The type parameter for an expression of type character is the length parameter.

7.1.4.1 Data Type, Type Parameters, and Shape of a Primary. The data type, type parameters, and shape of a primary are determined according to whether the primary is a constant, variable, function reference, or parenthesized expression. If a primary is a constant, its type and type parameters are determined by the constant (4.3). If it is a derived-type constructor, its type, type parameters, and shape are determined by the constructor name (4.4.2); if it is an array constructor, its type, type parameters, and shape are given in 4.5. If it is a variable or function reference, its type, type parameters, and shape are determined by the declaration for the variable (5.1) or by the name of the function subprogram (12.5.2.2), respectively. Note that in the case of a function reference, the function may be generic (13.9) or overloaded (12.5.4), in which case its type, type parameters, and shape are determined by the types, type parameters, and shapes of its actual arguments. If a primary is a parenthesized expression, its type, type parameters, and shape are those of the expression.

7.1.4.2 Data Type, Type Parameters, and Shape of the Result of an Operation. The type of an expression [x_1] *op* x_2 where *op* is an intrinsic operator is specified by Table 7.1. The data type of an expression [x_1] *op* x_2 where *op* is a defined operator is specified by the function subprogram defining the operation (7.3).

An expression whose type is real, double precision, complex, or character has type parameters. For an expression *op* x_2 where *op* is a numeric intrinsic unary operator and x_2 is of type real, double precision, or complex, the type parameters of the expression are those of the operand. For an expression x_1 *op* x_2 where *op* is a numeric intrinsic binary operator with one operand of type integer and the other of type real, double precision, or complex, the type parameters of the expression are those of the real, double precision, or complex operand. In the case where both operands are any of type real, double precision, or complex with type parameters p_1, r_1 and p_2, r_2 where the p's are precision parameter values and the r's are range parameter values, the type parameters of the expression are $max(p_1,p_2)$ and $max(r_1,r_2)$, respectively. For an expression x_1 // x_2 where // is the intrinsic operator for character concatenation, the type parameter is the sum of the lengths of the operands.

The shape of an expression [x_1] *op* x_2, where *op* is an intrinsic operator, is the shape of x_2, if *op* is unary or x_1 is scalar, and the shape of x_1, otherwise.

7.1.5 Conformability Rules for Intrinsic Operations. Two entities are in **shape conformance** if both are arrays of the same shape, or both are scalars, or one is an array and the other is a scalar.

For all intrinsic binary operations, the two operands must be in shape conformance. In case one is a scalar and the other an array, the scalar is treated as if it were an array of the same shape as the array operand with every element of the array equal to the value of the scalar.

7.1.6 Kinds of Expressions. An expression is either a scalar expression or an array expression.

7.1.6.1 Constant Expression. A **constant expression** is an expression in which each operator is an intrinsic operator, and each primary is:

(1) A constant,

(2) An array constructor where each element is a constant expression,

(3) A derived-type constructor where each component is a constant expression,

(4) An intrinsic function reference where each argument is a constant expression,

(5) An inquiry function reference where each argument is either a constant expression or a variable whose type parameters or bounds inquired about are not

assumed or allocated, or

(6) A constant expression enclosed in parentheses.

A **numeric constant expression** is a constant expression whose type is integer, real, double precision, or complex. An **integer constant expression** is a numeric constant expression whose type is integer. A **character constant expression** is a constant expression whose type is character. A **logical constant expression** is a constant expression whose type is logical.

The following are examples of constant expressions:

```
3
-3+4
SQRT (9.0)
'AB'
'AB' // 'CD'
('AB' // 'CD') // 'EF'
SIZE (A)
DIGITS (X) + 4
```

where A is an explicit-shaped array and X is of type default real.

7.1.6.2 Specification Expression. A **restricted expression** is an expression in that each primary is:

(1) A constant,

(2) A variable that is a dummy argument,

(3) A variable that is in a common block,

(4) A variable that is made accessible by a USE statement,

(5) An array constructor where each element is a restricted expression,

(6) A derived-type constructor where each component is a restricted expression,

(7) An intrinsic function reference where each argument is a restricted expression, or

(8) A restricted expression enclosed in parentheses.

A **specification expression** is a restricted expression which is scalar and of type integer.

The following are examples of specification expressions:

```
DLBOUND (B, 1) + 5
M + LEN (C)
```

where B, M, and C are dummy arguments and B is an assumed-shape array.

7.1.7 Evaluation of Operations. This section applies to both intrinsic and defined operations.

Any variable or function reference used 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. All of the characters in a character data object reference must be defined.

When a reference to a whole array or an array section is made, all of the selected elements must be defined. When a data object of a derived type is referenced, all of the nonvariant and selected variant components must be defined.

Any numeric 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 of type real or double precision to a real or double precision power is also prohibited.

The execution of a function reference must not alter the value of any other variable within the statement in which the function reference appears. The execution of a function reference in a statement must not alter the value of any other variable in common (5.4.2) or made accessible by a USE statement (11.3.1) if it affects the value of any other function reference in the statement. However, execution of a function reference in the logical expression of an IF statement (8.1.2.4) or WHERE statement (7.5.2.1) is permitted to affect variables in the statement that is executed when the value of the expression is true. For example, in the statements:

```
IF (F (X)) A = X
WHERE (G (X)) B = X
```

F or G may define X. 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 type of an expression in which a function reference appears does not affect the evaluation of the actual arguments of the function. The type of an expression in which a function reference appears does not affect and is not affected by the evaluation of the actual arguments of the function, except that the result of a function may assume a type that depends on the type of its arguments as specified in Sections 12 and 13.

Execution of an array element reference requires the evaluation of its subscripts. The type of an expression in which a subscript appears does not affect, and is not affected by, the evaluation of the subscript.

Execution of a substring reference requires the evaluation of its substring range. The type of an expression in which a substring name appears does not affect, and is not affected by, the evaluation of the substring expressions.

Execution of an array section reference requires the evaluation of its section subscripts. It is not necessary for a processor to evaluate any subscripts of a zero-sized array. The type of an expression in which an array section appears does not affect, and is not affected by, the evaluation of the array section subscripts.

When an intrinsic binary operator operates on a pair of operands and at least one of the operands is an array operand, the operation is performed element-by-element on corresponding array elements of the 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.

When an intrinsic unary operator operates on a single array operand, the operation is performed element-by-element, in any order, and the result is the same shape as the operand.

7.1.7.1 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 expression

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

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

```
X + W (Z)
```

where X is of size zero and W is a function, the function reference W(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 preceding examples, evaluation of these expressions causes Z to become undefined if L or W defines its argument.

7.1.7.2 Integrity of Parentheses. The sections that follow state certain conditions under which a processor may evaluate an expression different from the one specified by applying the rules given in 7.1.1, 7.2, and 7.3. However, any expression contained in parentheses must be treated as a data entity. For example, in evaluating the expression A + (B − C) where A, B and C are of numeric types, the difference of B and C must be evaluated before the addition operation is performed; the processor must not evaluate the mathematically equivalent expression (A + B) − C.

7.1.7.3 Evaluation of Numeric Intrinsic Operations. The rules given in 7.2.1 specify the interpretation of a numeric intrinsic operation. 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 expressions of a numeric type are mathematically equivalent if, for all possible values of their primaries, their mathematical values are equal. However, mathematically equivalent expressions of type numeric may produce different computational results. For example, any difference between the values of the expressions (1./3.)*3. and 1. is a computational difference, not a mathematical difference.

The mathematical definition of integer division is given in 7.2.1.1. The difference between the values of the expressions 5/2 and 5./2. is a mathematical difference, not a computational difference.

The following are examples of expressions 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 operands of numeric type.

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*A/Z	X*(A/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 with forbidden alternative forms that must not be used by a processor in the evaluation of those expressions.

Expression	Nonallowable 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 of type integer.

Each operand in a numeric intrinsic operation has a data type that may depend on the order of evaluation used by the processor. For example, in the evaluation of the expression

```
Z + R + I
```

where Z, R, and I represent terms of complex, real, and integer data type, respectively, the data type of the operand that is added to I may be either complex or real, depending on which pair of operands (Z and R, R and I, or Z and I) is added first.

7.1.7.4 Evaluation of Bit Intrinsic Operations. The rules given in 7.2.2 specify the interpretation of bit intrinsic operations. Once the interpretation of an expression has been established in accordance with those rules, the processor may evaluate any other expression that is bit-wise equivalent, provided that the integrity of parentheses is not violated. For example, for variables B1, B2, and B3 of type bit, the processor may choose to evaluate the expression

```
B1 .BOR. B2 .BOR. B3
```

as

```
B1 .BOR. (B2 .BOR. B3)
```

Two expressions of type bit are bit-wise equivalent if their values are equal for all possible values of their primaries.

7.1.7.5 Evaluation of the Character Intrinsic Operation. The rules given in 7.2.3 specify the interpretation of a character intrinsic operation. A processor needs to evaluate only as much of the character intrinsic operation as is required by the context in which the expression appears. For example, the statements

```
CHARACTER (LEN = 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.

7.1.7.6 Evaluation of Relational Intrinsic Operations. The rules given in 7.2.4 specify the interpretation of relational intrinsic operations. 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 expression

```
I .GT. J
```

where I and J are integer variables, as

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

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

7.1.7.7 Evaluation of Logical Intrinsic Operations. The rules given in 7.2.5 specify the interpretation of logical intrinsic operations. 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, for the variables L1, L2, and L3 of type logical, the processor may choose to evaluate the expression

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

as

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

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

7.1.7.8 Evaluation of a Defined Operation. The rules given in 7.3 specify the interpretation of a defined operation. Once the interpretation of an expression has been established in accordance with those rules, the processor may evaluate any other expression that is equivalent, provided that the integrity of parentheses is not violated.

Two expressions of derived-type are equivalent if their values are equal for all possible values of their primaries.

7.2 Interpretation of Intrinsic Operations.

The intrinsic operations are those defined in 7.1.2. These operations are divided into the following categories: numeric, bit, character, relational, and logical. The interpretations defined in the following sections apply to both scalars and arrays; for arrays, the interpretation for scalars is applied element-by-element.

The type, type parameters, shape, and interpretation of an expression that consists of an operator operating on a single operand or a pair of operands are independent of the context in which the expression appears. In particular, the type, type parameters, shape, and interpretation of such an expression are independent of the type, type parameters, and shape of any other 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 intrinsic conversion function, the expression INT (X + J) is an integer expression and X + J is a real expression.

7.2.1 Numeric Intrinsic Operations. A numeric operation is used to express a numeric computation. Evaluation of a numeric operation produces a numeric value. The permitted data types and shapes for operands of the numeric intrinsic operations are specified in 7.1.2. The permitted type parameters for operands of the numeric intrinsic operations are those that yield type parameters (7.1.4) of an approximation method supported by the processor.

The numeric operators and their interpretation in an expression are given in Table 7.2, where x_1 denotes the operand to the left of the operator and x_2 denotes the operand to the right of the operator.

Table 7.2. Interpretation of the Numeric Intrinsic Operators.

Operator	Representing	Use of Operator	Interpretation
**	Exponentiation	x_1 ** x_2	Raise x_1 to the power x_2
/	Division	x_1 / x_2	Divide x_1 by x_2
*	Multiplication	x_1 * x_2	Multiply x_1 by x_2
−	Subtraction	$x_1 - x_2$	Subtract x_2 from x_1
−	Negation	$- x_2$	Negate x_2
+	Addition	$x_1 + x_2$	Add x_1 and x_2
+	Identity	$+ x_2$	Same as x_2

The interpretation of a division may depend on the data types of the operands (7.2.1.1).

If M_1 and M_2 are of type integer and M_2 has a negative value, the interpretation of M_1 ** M_2 is the same as the interpretation of 1/(M_1 ** ABS(M_2)), which is subject to the rules of integer division (7.2.1.1). For example, 2**(−3) has the value of 1/(2**3), which is zero.

7.2.1.1 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 7.1 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 an operation is the integer closest to the mathematical quotient and between zero and the mathematical quotient inclusively. For example, the expression (−8)/3 has the value (−2).

7.2.1.2 Complex Exponentiation. In the case of a complex value raised to a complex power, the value of the operation is the "principal value" determined by x_1 ** x_2 = EXP(x_2 * LOG(x_1)), where EXP and LOG are functions described in 13.9.

7.2.2 Bit Intrinsic Operations. A bit operation is used to express a bit computation. Evaluation of a bit operation produces a result of type bit, with a value of B'0' or B'1'. The permitted data types and shapes for operands of the bit intrinsic operations are specified in 7.1.2.

The bit operators and their interpretation when used to form an expression are given in Table 7.3, where x_1 denotes the operand to the left of the operator and x_2 denotes the operand to the right of the operator.

Table 7.3. Interpretation of the Bit Intrinsic Operators.

Operator	Representing	Use of Operator	Interpretation
.BNOT.	Bit Negation	.BNOT. x_2	Bit negation of x_2
.BAND.	Bit Conjunction	x_1 .BAND. x_2	Bit conjunction of x_1 and x_2
.BOR.	Bit Inclusive Disjunction	x_1 .BOR. x_2	Bit inclusive disjunction of x_1 and x_2
.BXOR.	Bit Exclusive Disjunction	x_1 .BXOR. x_2	Bit exclusive disjunction of x_1 and x_2

The values of bit intrinsic operations are shown in Table 7.4.

Table 7.4. The Values of Operations Involving Bit Intrinsic Operators

x_1	x_2	.BNOT. x_2	x_1 .BAND. x_2	x_1 .BOR. x_2	x_1 .BXOR. x_2

B'1'	B'1'	B'0'	B'1'	B'1'	B'0'
B'1'	B'0'	B'1'	B'0'	B'1'	B'1'
B'0'	B'1'	B'0'	B'0'	B'1'	B'1'
B'0'	B'0'	B'1'	B'0'	B'0'	B'0'

7.2.3 Character Intrinsic Operation. The character intrinsic operator // is used to concatenate two operands of type character. Evaluation of the character intrinsic operation produces a result of type character. The permitted shapes for operands of the character intrinsic operation are specified in 7.1.2.

The interpretation of the character intrinsic operator // when used to form an expression is given in Table 7.5, where x_1 denotes the operand to the left of the operator and x_2 denotes the operand to the right of the operator.

Table 7.5. Interpretation of the Character Intrinsic Operator //.

Operator	Representing	Use of Operator	Interpretation
//	Concatenation	x_1 // x_2	Concatenate x_1 with x_2

The result of a character intrinsic operation is a character string whose value is the value of x_1 concatenated on the right with the value of x_2 and whose length is the sum of the lengths of x_1 and x_2. Parentheses used to specify the order of evaluation have no effect on the value of a character expression. For example, the value of ('AB' // 'CDE') // 'F' is the string 'ABCDEF'. Also, the value of 'AB' // ('CDE' // 'F') is the string 'ABCDEF'.

7.2.4 Relational Intrinsic Operations. A relational intrinsic operator is used to compare values of two operands using the relational intrinsic operators .LT., .LE., .GT., .GE., .EQ., .NE., <, <=, >, >=, ==, and <>. The permitted data types and shapes for operands of the relational intrinsic operators are specified in 7.1.2. Note, as shown in Table 7.1, that a relational intrinsic operator must not be used to compare the value of an expression whose type is numeric with one whose type is bit, character, or logical. Also, two operands of type logical must not be compared, and a complex operand can only be compared with another numeric operand when the operator is .EQ. .NE., ==, or <>.

Evaluation of a relational intrinsic operation produces a result of type logical, with a value of true or false.

The interpretation of the relational intrinsic operators is given in Table 7.6, where x_1 denotes the operand to the left of the operator and x_2 denotes the operand to the right of the operator. The operators <, <=, >, >=, ==, and <> have the same semantics as the operators .LT., .LE., .GT., .GE., .EQ., and .NE., respectively.

Table 7.6. Interpretation of the Relational Intrinsic Operators.

Operator	Representing	Use of Operator	Interpretation
.LT.	Less Than	x_1 .LT. x_2	x_1 less than x_2
<	Less Than	$x_1 < x_2$	x_1 less than x_2
.LE.	Less Than Or Equal To	x_1 .LE. x_2	x_1 less than or equal to x_2
<=	Less Than Or Equal To	$x_1 <= x_2$	x_1 less than or equal to x_2
.GT.	Greater Than	x_1 .GT. x_2	x_1 greater than x_2
>	Greater Than	$x_1 > x_2$	x_1 greater than x_2
.GE.	Greater Than Or Equal To	x_1 .GE. x_2	x_1 greater than or equal to x_2
>=	Greater Than Or Equal To	$x_1 >= x_2$	x_1 greater than or equal to x_2

.EQ.	Equal To	x_1 .EQ. x_2	x_1 equal to x_2
==	Equal To	x_1 == x_2	x_1 equal to x_2
.NE.	Not Equal To	x_1 .NE. x_2	x_1 not equal to x_2
<>	Not Equal To	x_1 <> x_2	x_1 not equal to x_2

A numeric relational intrinsic operation is interpreted as having the logical value true if the values of the operands satisfy the relation specified by the operator. A numeric relational intrinsic operation 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 numeric operands are of different types or have different type parameters but are in shape conformance, the value of the relational operation

x_1 *rel-op* x_2

is the value of the expression

$((x_1)-(x_2))$ *rel-op* 0

where 0 (zero) is of the same type, type parameters, and shape as the expression $((x_1)-(x_2))$, and *rel-op* is the same relational operator in both expressions.

A character relational intrinsic operation is interpreted as having the logical value true if the values of the operands satisfy the relation specified by the operator. A character relational intrinsic operation is interpreted as having the logical value false if the values of the operands do not satisfy the relation specified by the operator.

For a character relational intrinsic operation, the operands are compared one character at a time in order, beginning with the first character of each character operand. At the first position where the character operands differ, the character operand x_1 is considered to be less than x_2 if the character value of x_1 at this position precedes the value of x_2 in the collating sequence (3.1.4); x_1 is greater than x_2 if the character value of x_1 at this position follows the value of x_2 in the collating sequence. If the operands are of unequal length, the shorter operand is treated as if it were extended on the right with blanks to the length of the longer operand. Note that the collating sequence depends partially on the processor; however, the result of the use of the operators .EQ., .NE., ==, and <>. does not depend on the collating sequence.

A derived-type relational intrinsic operation is interpreted as having the logical value true if the values of the operands satisfy the relation specified by the operator. A derived-type relational intrinsic operation is interpreted as having the logical value false if the values of the operands do not satisfy the relation specified by the operator.

A derived-type operand x_1 is considered to be equal to x_2 if the values of all corresponding components (including variant components) of x_1 and x_2 are equal when of numeric, bit, character, or derived-type or equivalent (.EQV.), when of logical type. Otherwise, x_1 is considered to be not equal to x_2.

7.2.5 Logical Intrinsic Operations. A logical operation is used to express a logical computation. Evaluation of a logical operation produces a result of type logical, with a value of true or false. The permitted data types and shapes for operands of the logical intrinsic operations are specified in 7.1.2.

The logical operators and their interpretation when used to form an expression are given in Table 7.7, where x_1 denotes the operand to the left of the operator and x_2 denotes the operand to the right of the operator.

Table 7.7. Interpretation of the Logical Intrinsic Operators.

Use of

Operator	Representing	Operator	Interpretation
.NOT.	Logical Negation	.NOT. x_2	Logical negation of x_2
.AND.	Logical Conjunction	x_1 .AND. x_2	Logical conjunction of x_1 and x_2
.OR.	Logical Inclusive Disjunction	x_1 .OR. x_2	Logical inclusive disjunction of x_1 and x_2
.NEQV.	Logical Non-equivalence	x_1 .NEQV. x_2	Logical non-equivalence of x_1 and x_2
.EQV.	Logical Equivalence	x_1 .EQV. x_2	Logical equivalence of x_1 and x_2

The values of the logical intrinsic operations are shown in Table 7.8.

Table 7.8. The Values of Operations Involving Logical Intrinsic Operators

x_1	x_2	.NOT. x_2	x_1 .AND. x_2	x_1 .OR. x_2	x_1 .EQV. x_2	x_1 .NEQV. x_2
true	true	false	true	true	true	false
true	false	true	false	true	false	true
false	true	false	false	true	false	true
false	false	true	false	false	true	false

7.3 Interpretation of Defined Operations. The interpretation of a defined operation is provided by the function subprogram that defines the operation.

7.3.1 Unary Defined Operation. A function subprogram defines the unary operation *op* x_2 if:

(1) The function subprogram is specified with a FUNCTION statement of the form (12.5.2.2):

    [RECURSIVE] [*type-spec*] FUNCTION *function-name* (d_2) &
    [RESULT (*res*)] OPERATOR (*operator*)

(2) The interface to the function subprogram is explicit,

(3) The type of x_2 is the same as the type of dummy argument d_2,

(4) The type parameters, if any, of x_2 must match those of d_2, for those type parameters of d_2 not specified with an asterisk (*), and

(5) d_2 is a scalar and x_2 is a scalar or array, or d_2 and x_2 are arrays of the same shape.

7.3.2 Binary Defined Operation. A function subprogram defines the binary operation x_1 *op* x_2 if:

(1) The function subprogram is specified with a FUNCTION statement of the form (12.5.2.2):

    [RECURSIVE] [*type-spec*] FUNCTION *function-name* &
    (d_1, d_2) [RESULT (*res*)] OPERATOR (*operator*)

(2) The interface to the function subprogram is explicit,

(3) The types of x_1 and x_2 are the same as those of the dummy arguments d_1 and d_2, respectively,

(4) The type parameters, if any, of x_1 and x_2 must match those of d_1 and d_2, respectively, for those type parameters of d_1 and d_2 not specified with an asterisk (*), and

(5) d_1 and d_2 are scalar and x_1 and x_2 have the same shape, or d_1 or d_2 (or both) is an array and the shapes of x_1 and x_2 match those of d_1 and d_2, respectively.

7.4 Precedence of Operators. There is a precedence among the intrinsic and extension operations implied by the general form in 7.1.1, which determines the order in which the operands are combined, unless the order is changed by the use of parentheses. This precedence order is summarized in Table 7.9.

Table 7.9. Categories of Operations and Relative Precedences.

Category of Operation	Operators	Precedence
Extension	*defined-unary-op*	Highest
Numeric	**	.
Numeric	* or /	.
Numeric	unary + or −	.
Numeric	binary + or −	.
Bit	.BNOT.	.
Bit	.BAND.	.
Bit	.BOR. or .BXOR.	.
Character	//	.
Relational	.EQ., .NE., .LT., .LE., .GT., .GE.	.
	= =, < >, <, < =, >, > =	
Logical	.NOT.	.
Logical	.AND.	.
Logical	.OR.	.
Logical	.EQV. or .NEQV.	.
Extension	*defined-binary-op*	Lowest

Overloaded operations have the precedence of the intrinsic or extension operation associated with the operator with the same symbol name.

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 general form of an expression (7.1.1) also establishes a precedence among operators in the same syntactic class. This precedence determines the order in which the operands are to be combined unless the order is changed by the use of parentheses. For example, in interpreting a *level-2-expr* containing two or more binary operators + or −, each operation (*add-operand*) is combined from left to right. Similarly, the same left to right interpretation for a *mult-operand* in *add-operand*, or *level-3-expr* in *level-4-expr*, as well as for other kinds of expressions, is a consequence of the general form (7.1.1). However, for interpreting a *mult-operand* expression when two or more exponential operators ** combine *level-1-expr* operands, each *level-1-expr* is combined from right to left. For example, the expressions

```
2.1 + 3.4 + 4.9
2.1 * 3.4 * 4.9
2.1 / 3.4 / 4.9
2 ** 3 ** 4
```

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

have the same interpretations as the expressions

```
(2.1 + 3.4) + 4.9
(2.1 * 3.4) * 4.9
(2.1 / 3.4) / 4.9
2 ** (3 ** 4)
('AB' // 'CD') // 'EF'
```

Note that as a consequence of the general form (7.1.1), only the first *add-operand* of a *level-2-expr* may be preceded by the identity (+) or negation (−) operator. Note also that these formation rules do not permit expressions containing two consecutive numeric operators, such as A ** −B or A + −B. However, expressions such as A ** (−B) and A + (−B) are permitted.

As another example, in the expression

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

the general form (7.1.1) implies a higher precedence for the .AND. operator 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)
```

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 a numeric operator, a relational operator, and a logical operator. This expression would be interpreted the same as the expression

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

7.5 Assignment.

7.5 Assignment. Execution of an assignment causes a variable to become defined or redefined.

An assignment is either an assignment statement, a masked array assignment, or an element array assignment.

7.5.1 Assignment Statement. Any variable may be defined or redefined by execution of an assignment statement.

7.5.1.1 General Form.

R716 *assignment-stmt* **is** *variable* = *expr*

where *variable* is defined in 2.4.4 and *expr* is defined in 7.1.1.8.

variable must not include an array element more than once in an array section with vector subscripts or in an identified array.

Examples of an assignment statement are:

```
A = 3.5 + X * Y
I = INT (A)
```

An assignment statement is either intrinsic or defined.

7.5.1.2 Intrinsic Assignment Statement. An **intrinsic assignment statement** is an assignment statement where:

(1) The types of *variable* and *expr* are intrinsic, as specified in Table 7.10 for assignment, or

(2) The types of *variable* and *expr* are of the same derived type.

A **numeric intrinsic assignment statement** is an intrinsic assignment statement for which *variable* and *expr* are of numeric type. A **character intrinsic assignment statement** is an intrinsic assignment statement for which *variable* and *expr* are of type character. An **array intrinsic assignment statement** is an intrinsic assignment statement for which *variable* is an array. A **logical intrinsic assignment statement** is an intrinsic assignment statement for which *variable* and *expr* are of type logical. A **bit intrinsic assignment statement** is an intrinsic assignment statement for which *variable* and *expr* are of type bit. A **derived-type intrinsic assignment statement** is an intrinsic assignment statement for which *variable* and *expr* are of the same derived type.

Table 7.10. Type Conformance for the Assignment Statement *variable* = *expr*

Type of *variable*	Type of *expr*
integer	integer, real, double precision, complex
real	integer, real, double precision, complex
double precision	integer, real, double precision, complex
complex	integer, real, double precision, complex
bit	bit
character	character
logical	logical
derived type	same derived type as *variable*

7.5.1.3 Defined Assignment Statement. A **defined assignment statement** is an assignment statement which is not an intrinsic assignment statement, and for which there exists an accessible subroutine subprogram that defines the assignment.

7.5.1.4 Intrinsic Assignment Conformance Rules. For the intrinsic assignment statement, *variable* and *expr* must be in shape conformance, and if *expr* is an array, *variable* must also be an array. The types of *variable* and *expr* must conform with the rules of Table 7.10.

For a numeric intrinsic assignment statement, *variable* and *expr* may have different numeric types or different type parameters, in which case the value of *expr* is converted to the type and type parameters of *variable* according to the rules of Table 7.11.

Table 7.11. Numeric Conversion and Assignment Statement of *expr* to *variable* (The functions INT, REAL, DBLE and CMPLX are the generic functions defined in 13.9.)

Type of *variable*	Value Assigned
integer	INT(*expr*)
real	REAL(*expr*, MOLD = *variable*)
double precision	DBLE(*expr*)
complex	CMPLX(*expr*, MOLD = *variable*)

For a character intrinsic assignment statement, *variable* and *expr* may have different type parameters (lengths) in which case the conversion of *expr* to the length of *variable* is:

(1) If the length of *variable* is less than that of *expr*, the value of *expr* is truncated from the right until it is the same length as *variable*;

(2) If the length of *variable* is greater than that of *expr*, the value of *expr* is extended to the right with blanks until it is the same length as *variable*.

7.5.1.5 Interpretation of Intrinsic Assignments. Execution of an intrinsic assignment causes, in effect, the evaluation of the expression *expr* and all expressions within *variable* (7.1.7), the possible conversion of *expr* to the type and type parameters of *variable* (Table 7.11), and the definition of *variable* with the resulting value. The execution of the assignment must appear as if the evaluation of all operations in *expr* and, if present, all operations in the subscripts or section subscripts of *variable* occurred before any portion of *variable* is defined by the assignment.

Both *variable* and *expr* may contain references to any portion of *variable*.

If *expr* in an assignment is a scalar and *variable* is an array, the *expr* is treated as if it were an array of the same shape as *variable* with every element of the array equal to the scalar value of *expr*.

When *variable* in an intrinsic assignment is an array, the assignment is performed element-by-element on corresponding array elements of *variable* and *expr*. For example, where A and B are arrays of the same shape, the array intrinsic assignment

```
A = B
```

assigns the corresponding elements of B to those of A; that is, the first element of B is assigned to the first element of A, the second element of B is assigned to the second element of A, etc. The processor may perform the element-by-element assignment in any order.

When *variable* is an array section, the assignment does not affect the definition status or value of the elements of the parent array not specified by the array section.

7.5.1.6 Interpretation of Defined Assignment Statements. The interpretation of a defined assignment is provided by the subroutine subprogram that defines the operation.

A subroutine subprogram defines the defined assignment $x_1 = x_2$ if:

(1) The subroutine subprogram is specified with a SUBROUTINE statement of the form (12.5.2.3):

SUBROUTINE *subroutine-name* (d_1, d_2) ASSIGNMENT

(2) The interface to the subroutine subprogram is explicit,

(3) The types of x_1 and x_2 are the same as those of the dummy arguments d_1 and d_2, respectively,

(4) The type parameters, if any, of x_1 and x_2 must match those of d_1 and d_2, respectively, for those type parameters of d_1 and d_2 not specified with an asterisk (*), and

(5) d_1 and d_2 are scalar and x_1 and x_2 have the same shape, or d_1 or d_2 (or both) is an array and the shapes of x_1 and x_2 match those of d_1 and d_2, respectively.

7.5.2 Masked Array Assignment (WHERE). The masked array assignment is used to mask the evaluation of expressions and assignment of values in array assignment statements, according to the value of a logical or bit array expression.

7.5.2.1 General Form of the Masked Array Assignment. A **masked array assignment** is either a WHERE statement or WHERE construct.

```
R717  where-stmt              is WHERE ( array-mask-expr )  array-assignment-stmt

R718  where-construct         is where-construct-stmt
                                        [ array-assignment-stmt ]...
                                     [ elsewhere-stmt
                                        [ array-assignment-stmt ]... ]
                                     end-where-stmt

R719  where-construct-stmt    is WHERE ( array-mask-expr )

R720  array-mask-expr         is logical-expr
                              or bit-expr

R721  elsewhere-stmt          is ELSEWHERE

R722  end-where-stmt          is END WHERE
```

Constraint: The shape of the *array-mask-expr* and the variable being defined in each *array-assignment-stmt* must be the same.

Examples of a masked array assignment are:

```
WHERE (TEMP > 100.0) TEMP = TEMP - REDUCE_TEMP

WHERE (PRESSURE <= 1.0)
   PRESSURE = PRESSURE + INC_PRESSURE
   TEMP = TEMP - 5.0
END WHERE
```

7.5.2.2 Interpretation of Masked Array Assignments. The execution of a masked array assignment causes the expression *array-mask-expr* to be evaluated. The array assignment statements following the WHERE and ELSEWHERE keywords are executed in normal execution sequence. An array may be defined in more than one array assignment statement in a WHERE construct. A reference to an array may appear subsequent to its definition in the same WHERE construct.

When an *array-assignment* is executed in a *masked-array-assignment*, the *expr* in the *where-stmt* or each *expr* in the array assignment statements, immediately following the WHERE keyword, is evaluated for all elements where *array-mask-expr* is true (or for all elements where *array-mask-expr* is false in the array assignment statements following ELSEWHERE), and the result is assigned to the corresponding elements of *variable*. For each false value of *array-mask-expr* (or true value for the array assignment statements after ELSEWHERE) the value of the corresponding element of *variable* in each array assignment statement immediately following the WHERE keyword is not affected, and it is as if the expression *expr* were not evaluated. If an *array-mask-expr* is of type BIT, the elements with value B'1' are treated as true and elements with value B'0' are treated as false.

If a transformational function reference occurs in *expr*, it is evaluated without any masked control by the *array-mask-expr*; that is, all of its argument expressions are fully evaluated and the function is fully evaluated. Elements corresponding to true values in *array-mask-expr* (false in the *expr* after ELSEWHERE) are selected for use in evaluating each *expr*.

In a masked array assignment, only a WHERE statement may be a branch target. Changes to entities in *array-mask-expr* will not affect the execution of statements in the *masked-array-assignment*. Execution of an END WHERE has no effect.

7.5.3 Element Array Assignment (FORALL). The element array assignment statement is used to specify an array assignment in terms of array elements or array sections. The element array assignment may be masked with a scalar logical or bit expression.

7.5.3.1 General Form of Element Array Assignment.

R723	*forall-stmt*	**is**	FORALL (*forall-triplet-spec-list* [,*scalar-mask-expr*]) *forall-assignment*
R724	*forall-triplet-spec*	**is**	*subscript-name* = *subscript* : *subscript* [: *stride*]

Constraint: *subscript-name* must be a *scalar-symbolic-name* of type integer.

Constraint: The subscripts and stride in a *forall-triplet-spec* must not contain any references to any *subscript-name* in the *forall-triplet-spec-list*.

R725	*forall-assignment*	**is**	*array-element* = *expr*
		or	*array-section* = *expr*

Constraint: The *array-section* or *array-element* must contain references to all subscript names in the *forall-triplet-spec-list*. *expr* in a *forall-assignment* must reference all of the *forall-triplet-spec subscript-names*.

For each subscript name in the *forall-assignment*, the set of permitted values is determined on entry to the statement and is

$$m_1 + (k-1) \times m_3, \text{ where } k = 1, 2, ..., \mathrm{INT}((m_2 - m_1 + m_3)/m_3)$$

and where m_1, m_2, and m_3 are the values of the first subscript, the second subscript, and the stride respectively in the *forall-triplet-spec*. If *stride* is missing, it is as if it were present with a value of the integer 1. The expression *stride* must not have the value 0. If for some subscript name $INT((m_2 - m_1 + m_3)/m_3) \leq 0$, the *forall-assignment* is not executed.

Examples of element array assignments are:

```
FORALL (I = 1:N, J = 1:N) H (I, J) = 1.0 / REAL (I + J - 1)

FORALL (I = 1:N, J = 1:N, A (I, J) .NE. 0.0) B (I, J) = 1.0 / A (I, J)
```

7.5.3.2 Interpretation of Element Array Assignments. Execution of an element array assignment consists of the evaluation in any order of the subscript and stride expressions in the *forall-triplet-spec-list*, the evaluation of the scalar mask expression, and the evaluation of the *expr* in the *forall-assignment* for all valid combinations of subscript names for which the scalar mask expression is true, followed by the assignment of these values to the corresponding elements of the array being assigned to. If the scalar mask expression is omitted, it is as if it were present with value true. If the scalar mask expression is of type BIT, an expression with value B'1' is treated as true and an expression value B'0' is treated as false.

The *forall-assignment* must not cause any element of the array being assigned to be assigned a value more than once. The scope of the subscript name is the FORALL statement itself. A function reference appearing in any expression in the *forall-assignment* must not redefine any subscript name.

8 EXECUTION CONTROL

Control constructs are used to control the execution sequence. These constructs include executable constructs containing blocks and executable statements that may be used to alter the execution sequence.

8.1 Executable Constructs Containing Blocks. The following are executable constructs containing blocks that may be used to control the execution sequence:

(1) IF Construct

(2) CASE Construct

(3) DO Construct

(4) ENABLE Construct

A **block** is a sequence of executable constructs that is treated as an integral unit.

R801 *block* **is** [*execution-part*]...

Executable constructs may be used to control which blocks of a program are executed or how many times a block is executed. Blocks are always bounded by statements that are particular to the construct in which they are embedded. Note that a block may be empty.

Any of these four constructs may be named with a symbolic name. If a construct is named, the name must be the first lexical element of the first statement of the construct and the last lexical element of the construct. In fixed form, the preceding name must be placed after column 6.

There is a simplified form of the IF construct (the IF statement) that contains a single action statement.

8.1.1 Rules Governing Blocks.

8.1.1.1 Executable Constructs in Blocks. If a block contains an executable construct, the executable construct must be entirely contained within the block.

8.1.1.2 Control Flow in Blocks. Transfer of control to the interior of a block from outside the block is prohibited. Transfers within a block may occur. For example, if a statement inside the block has a statement label, a GO TO statement using that label may appear in the same block. CALL statements and function references may appear in a block (12.4.2,12.4.4).

8.1.1.3 Execution of a Block. Execution of a block begins with the execution of the first executable construct in the block. Unless there is a transfer of control out of the block, the execution of the block is completed when the last executable construct in the sequence is executed. The action that takes place at the terminal boundary depends on the position of the block within a particular construct. It is usually a transfer of control.

8.1.2 IF Construct.

The **IF construct** selects for execution no more than one of its constituent blocks. The **IF statement** controls the execution of a single statement.

8.1.2.1 Form of the IF Construct.

```
R802  if-construct          is  if-then-stmt
                                    block
                                  [ else-if-stmt
```

block]...
[*else-stmt*
block]
end-if-stmt

R803	*if-then-stmt*	**is**	[*if-construct-name* :] IF (*scalar-mask-expr*) THEN
R804	*else-if-stmt*	**is**	ELSE IF (*scalar-mask-expr*) THEN
R805	*else-stmt*	**is**	ELSE
R806	*end-if-stmt*	**is**	END IF [*if-construct-name*]

Constraint: If an *if-construct-name* is present, the same name must be specified on both the *if-then-stmt* and the corresponding *end-if-stmt.*

8.1.2.2 Execution of an IF Construct. At most one of the blocks contained within the IF construct is executed. If there is an ELSE statement in the construct, exactly one of the blocks contained within the construct will be executed. The block that is executed is the one following the first scalar mask expression that evaluates to the value .TRUE. If none of the scalar mask expressions evaluate to the value .TRUE., the block following the ELSE statement, if any, is executed. The scalar mask expressions are evaluated in the order of their appearance in the construct until a true value is found or an ELSE statement or END IF statement is encountered. If a true value or an ELSE statement is found, the block immediately following is executed and this completes the execution of the construct. The expressions in any remaining ELSE IF statements of the IF construct are not evaluated.

If none of the evaluated expressions are true and there is no ELSE statement, the execution of the construct is completed without the execution of any blocks within the construct.

If the scalar mask expression if of type BIT, an expression with value B'1' is treated as true and an expression with value B'0' is treated as false.

An ELSE IF statement or an ELSE statement must not be a branch target. It is permissible to branch to an END IF statement from within the IF construct, and also from outside the construct.

8.1.2.3 Examples of IF Constructs.

```
IF (CVAR .EQ. 'RESET') THEN
   I = 0; J = 0; K = 0
END IF

IF (PROP) THEN
   WRITE (3, '("QED")')
   STOP
ELSE
   PROP = NEXTPROP
END IF

IF (A .GT. 0) THEN
   B = C/A
   IF (B .GT. 0) THEN
      D = 1.0
   END IF
ELSE IF (C .GT. 0) THEN
   B = A/C
   D = -1.0
ELSE
   B = ABS (MAX (A, C))
   D = 0
END IF
```

8.1.2.4 IF Statement. The IF statement controls a single action statement (R218).

R807	*if-stmt*	**is** IF (*scalar-mask-expr*) *action-stmt*

Constraint: The *action-stmt* in the *if-stmt* must not be an *if-stmt*.

Execution of an IF statement causes evaluation of the scalar mask expression. If the value of the expression is true, the action statement is executed. If the value is false, the action statement is not executed and execution continues as though a CONTINUE statement were executed.

If the scalar mask expression if of type BIT, an expression with value B'1' is treated as true and an expression with value B'0' is treated as false.

The execution of a function reference in the scalar mask expression is permitted to affect entities in the action statement.

8.1.3 CASE Construct. The **CASE construct** selects for execution exactly one of its constituent blocks.

8.1.3.1 Form of the CASE Construct.

R808	*case-construct*	**is** *select-case-stmt*
		[*case-stmt*
		block]...
		end-select-stmt
R809	*select-case-stmt*	**is** [*select-construct-name* :] SELECT CASE (*case-expr*)
R810	*case-stmt*	**is** CASE *case-selector*
R811	*end-select-stmt*	**is** END SELECT [*select-construct-name*]

Constraint: If a *select-construct-name* is present, the same name must be specified on both the *select-case-stmt* and the corresponding *end-select-stmt*.

R812	*case-expr*	**is** *scalar-int-expr*
		or *scalar-char-expr*
		or *scalar-logical-expr*
		or *scalar-bit-expr*
R813	*case-selector*	**is** (*case-value-range-list*)
		or DEFAULT

Constraint: Only one DEFAULT *case-selector* may appear in any given *case-construct*.

R814	*case-value-range*	**is** *case-value*
		or [*case-value*] : [*case-value*]
R815	*case-value*	**is** *scalar-int-constant-expr*
		or *scalar-char-constant-expr*
		or *scalar-logical-constant-expr*
		or *scalar-bit-constant-expr*

The case selector may specify a value or a range. For a given CASE construct, the case value in the case selectors must be of the same type as the case expression that appears in the SELECT CASE statement. For character type, length differences are allowed.

8.1.3.2 Execution of a CASE Construct. The execution of the SELECT CASE statement causes the case expression to be evaluated. The resulting value is called the **case index** and must match one of the selectors of one of the CASE statements of the construct. For a case value range list, a match occurs if the case index matches any of the case-value ranges in the list. For a case index with a value of *c*, a match is determined as follows:

(1) If the case value range contains a single value *v* without a colon, a match occurs for data type logical if the expression *c* .EQV. *v* is true. A match occurs for data type integer, character, or bit if the expression *c* .EQ. v is true.

(2) If the case value range is of the form *low* : *high*, the data type must not be logical and a match occurs if the expression *low* .LE. *c* .AND. *c* .LE. *high* is true.

(3) If the case value range is of the form *low* :, the data type must not be logical and a match occurs if the expression *low* .LE. *c* is true.

(4) If the case value range is of the form : *high*, the data type must not be logical and a match occurs if the expression *c* .LE. *high* is true.

(5) If the case-value range is of the form :, the data must not be logical or bit and a match always occurs. A case construct containing such a case selector must not contain any other case selector except possibly a DEFAULT selector.

(6) If no other selector matches, a DEFAULT selector must be present, and it matches the case index.

The case value ranges in different selectors must not overlap; that is, there must be no possible value of the data type that matches more than one selector. Case-value ranges within a single case selector may overlap.

The block following the CASE statement containing the matching selector is executed. This completes execution of the construct.

One and only one of the blocks of a CASE construct is executed.

A CASE statement must not be a branch target. It is permissible to branch to an END SELECT statement only from within the CASE construct.

8.1.3.3 Examples of CASE Constructs. An integer signum function:

```
INTEGER FUNCTION SIGNUM (N)
SELECT CASE (N)
CASE (:-1)
   SIGNUM = -1
CASE (0)
   SIGNUM = 0
CASE (1:)
   SIGNUM = 1
END SELECT
END
```

A code fragment to check for balanced parentheses:

```
CHARACTER LINE (80)
...
LEVEL=0
DO I = 1, 80
   SELECT CASE (LINE(I:I))
   CASE ('(')
      LEVEL = LEVEL + 1
   CASE (')')
```

```
         LEVEL = LEVEL - 1
         IF (LEVEL .LT. 0) THEN
            PRINT *, 'UNEXPECTED RIGHT PARENTHESIS'
            EXIT
         END IF
      CASE DEFAULT
         !IGNORE ALL OTHER CHARACTERS
      END SELECT
   END DO
   IF (LEVEL .GT. 0) THEN
      PRINT *, 'MISSING RIGHT PARENTHESIS'
   END IF
```

The following three fragments are equivalent:

```
IF (SILLY .EQ. 1) THEN
   CALL THIS
ELSE
   CALL THAT
END IF

SELECT CASE (SILLY .EQ. 1)
CASE (.TRUE.)
   CALL THIS
CASE (.FALSE.)
   CALL THAT
END SELECT

SELECT CASE (SILLY .EQ. 1)
CASE DEFAULT
   CALL THAT
CASE (.TRUE.)
   CALL THIS
END SELECT
```

8.1.4 Iteration Control. The DO construct is used to provide iteration control by specifying the repeated execution of a sequence of executable constructs.

8.1.4.1 Form of the DO Construct.

R816	*do-construct*	**is** *do-stmt*
		do-body
		do-termination
R817	*do-stmt*	**is** [*do-construct-name* :] DO [*label*] [[,] *loop-control*]
R818	*loop-control*	**is** *do-variable* = *scalar-numeric-expr*, *scalar-numeric-expr* [, *scalar-numeric-e*
		or (*scalar-int-expr* TIMES)

Constraint: The *do-variable* must be a scalar integer, real, or double precision variable.

R819	*do-body*	**is** [*execution-part*]...
R820	*do-termination*	**is** *end-do-stmt*
		or *continue-stmt*
		or *do-term-stmt*
		or *do-construct*

R821 *do-term-stmt* **is** *action-stmt*

Constraint: If the *label* is omitted in a *do-stmt*, the corresponding *do-termination* must be an *end-do-stmt*.

Constraint: If a *label* appears in the *do-stmt* and the corresponding *do-termination* is not a *do-construct*, the *do-termination* must be identified with that label.

Constraint: If the *do-termination* is a *continue-stmt* or *do-term-stmt*, the corresponding *do-stmt* must contain a label.

Constraint: A *do-term-stmt* must not be a *continue-stmt*, *goto-stmt*, *return-stmt*, *stop-stmt*, *exit-stmt*, *cycle-stmt*, *arithmetic-if-stmt*, *assigned-goto-stmt*, *computed-goto-stmt*, nor an *if-stmt* that causes a transfer of control.

Constraint: If the *do-termination* is a *do-construct*, both of the corresponding *do-stmt*s must specify the same label.

Constraint: If a *do-termination* is a *do-construct*, the *do-termination* of that *do-construct* must not be an *end-do-stmt*.

R822 *end-do-stmt* **is** END DO [*do-construct-name*]

Constraint: If a *do-construct-name* is used on the *do-stmt*, the corresponding *do-termination* must be an *end-do-stmt* that uses the same *do-construct-name*. If a *do-construct-name* does not appear on the *do-stmt*, a *do-construct-name* must not appear on the corresponding *do-termination*.

R823 *exit-stmt* **is** EXIT [*do-construct-name*]

R824 *cycle-stmt* **is** CYCLE [*do-construct-name*]

An EXIT statement of CYCLE statement is said to **belong** to a specific DO construct. If the EXIT statement or CYCLE statement contains a construct name, it belongs to the DO construct using that name. Otherwise, it belongs to the innermost DO construct in which it appears.

8.1.4.2 Range of a DO Construct. The **range** of a DO construct consists of the *do-body* and the *continue-stmt*, *do-term-stmt*, or termination *do-construct*, if any. The range must satisfy the rules for blocks (8.1.1).

Within a program unit, all DO constructs whose DO statements use the same label are said to share the statement identified with that label. Note that the statement so identified must be a CONTINUE statement or *do-term-stmt* that serves as the do termination of the innermost of these DO constructs.

Note that if the *do-termination* is an END DO statement, the range is a block (8.1). If the *do-termination* is a *continue-stmt*, *do-term-stmt*, or *do-construct*, a terminal boundary delimiting the range is assumed (8.1.1.3).

8.1.4.3 Active and Inactive DO Constructs. A DO construct is either **active** or **inactive**. Initially inactive, a DO construct becomes active only when its DO statement is executed.

Once active, the DO construct becomes inactive only when the construct it specifies is terminated (8.1.4.4.4).

When a DO construct becomes inactive, the *do-variable*, if any, retains its last defined value.

8.1.4.4 Execution of a DO Construct. A DO construct specifies a loop. A **loop** is a sequence of executable constructs that is executed repeatedly. There are three phases in the execution of a DO construct: initiation of the loop, execution of the loop body, and termination of the loop.

8.1.4.4.1 Loop Initiation. When the DO statement is executed, the DO construct becomes active. If there is *loop-control* of the form *do-variable* = *num-expr*$_1$, *num-expr*$_2$ [, *num-expr*$_3$], the following steps are performed in sequence:

(1) The *initial parameter* m_1, the *terminal parameter* m_2, and the *incrementation parameter* m_3 are established by evaluating *num-expr*$_1$, *num-expr*$_2$, and *num-expr*$_3$, respectively, including, if necessary, conversion to the type of the *do-variable* according to the rules for numeric conversion (Table 7.10). If *num-expr*$_3$ does not appear, m_3 has a value of one. m_3 must not have a value of zero.

(2) The DO variable becomes defined with the value of the *initial-parameter* m_1.

(3) The **iteration count** is established and is the value of the expression

$$\text{MAX}(\ (m_2 - m_1 + m_3)\ /\ m_3,\ 0)$$

Note that the iteration count is zero whenever:

$m_1 > m_2$ and $m_3 > 0$, or
$m_1 < m_2$ and $m_3 < 0$.

If *loop-control* takes the form *scalar-int-expr* TIMES, the *scalar-int-expr* is evaluated. If the resulting value is nonnegative, it becomes the iteration count; otherwise, the iteration count is zero.

At the completion of the execution of the DO statement, the execution cycle begins.

8.1.4.4.2 The Execution Cycle. The **execution cycle** of a DO construct consists of the following steps performed in sequence:

(1) The iteration count, if any, is tested. If the iteration count is zero, the *do-construct* becomes inactive. If, as a result, all of the *do-constructs* sharing the *do-term-stmt* or *continue-stmt* are inactive, normal execution continues with execution of the next executable construct following the *do-term-stmt* or *continue-stmt*. However, if some of the DO constructs sharing the *do-term-stmt* or *continue-stmt* are active, execution continues with step (3) of the execution cycle of the active DO construct whose DO statement was most recently executed.

(2) If the iteration count is nonzero, the range of the DO construct is executed.

(3) The iteration count, if any, is decremented by one. The *do-variable*, if any, is incremented by the value of the incrementation parameter m_3.

(4) This cycle is executed repeatedly from step (1) until the loop is terminated.

Except for the incrementation of the DO variable that occurs in step (3), the DO variable must neither be redefined nor become undefined while the DO construct is active. Execution of the *do-termination* occurs as a result of the normal execution sequence or as a result of a transfer of control from within the range of the DO construct. Unless execution of the *do-term-stmt*, if any, results in a transfer of control, execution continues with step (3) of the execution cycle.

8.1.4.4.3 Cycle Interruption. Execution of a CYCLE statement that belongs to a DO construct causes immediate execution of step (3) of the current execution cycle of that DO construct.

8.1.4.4.4 Loop Termination. The execution of the loop is complete when one of the following conditions occurs:

(1) The iteration count, tested during step (1) of the execution cycle, is determined to be zero.

(2) An EXIT statement that belongs to this DO construct, or to a DO construct that contains this one, is executed.

(3) A CYCLE statement that belongs to a DO construct that contains this one is executed.

(4) A RETURN statement within the range is executed.

(5) Control is transferred outside the range by the execution of a statement that causes a transfer of control.

(6) A STOP statement anywhere in the program is executed, or execution is terminated for any other reason.

8.1.4.5 Examples of DO Constructs. Example 1:

```
DO
   IF (X .GT. Y) THEN
      Z = X
      EXIT
   END IF
   CALL NEWX (X)
END DO
```

Example 2:

```
SUM = 0
READ *, N
DO (N TIMES)
   READ *, P, Q
   CALL CALCULATE (P, Q, R)
   SUM = SUM + R
   IF (SUM .GT. SMAX) EXIT
END DO

N = 0
DO I = 1, 10
   J = I
   DO K = 1, 5
      L = K
      N = N + 1
   END DO
END DO
```

After execution of the above program fragment, I = 11, J = 10, K = 6, L = 5, and N = 50.

Example 3:

```
N = 0
DO I = 1, 10
   J = I
   DO K = 5, 1
      L = K
      N = N + 1
   END DO
END DO
```

After execution of the above program fragment, I=11, J=10, K=5, N=0. L is not defined.

Example 4:

```
      N = 0
      DO 100 I = 1, 10
      J = I
         DO 100 K = 1, 5
            L = K
100         N = N + 1
```

After execution of the above statements, I = 11, J = 10, K = 6, L = 5, and N = 50.

Example 5:

```
      N = 0
      DO 200 I = 1, 10
         J = I
         DO 200 K = 5, 1
            L = K
200         N = N + 1
```

After execution of the above statements I = 11, J = 10, K = 5, N = 0. L is not defined.

8.1.5 ENABLE Construct. The **ENABLE construct** permits control of the execution sequence when it is desirable to check for exceptional conditions. An exceptional condition represents an event that is either intrinsically defined or user defined. When enabled, a condition may be **signaled** (set on, explicitly or implicitly) when the associated event occurs. Signaling a condition causes a transfer of control to a sequence of statements called a **condition handler**.

8.1.5.1 Form of the ENABLE Construct.

R825	*enable-construct*	**is** *enable-stmt* *block* [*handle-stmt* *block*]... *end-enable-stmt*
R826	*enable-stmt*	**is** [*enable-construct-name* :] ENABLE [(*condition-name-list*)]
R827	*handle-stmt*	**is** HANDLE (*condition-name-list*) **or** HANDLE (*)
R828	*end-enable-stmt*	**is** END ENABLE [*enable-construct-name*]

Constraint: A *condition-name* must not appear more than once in a single *condition-name-list*.

Constraint: A *condition-name* appearing in an *enable-stmt* or *handle-stmt* must not be a dummy argument.

Constraint: HANDLE (*) may appear at most once in an ENABLE construct.

Constraint: If an *enable-construct-name* is present, the same name must be specified on both the *enable-stmt* and the corresponding *end-enable-stmt.*

The block immediately following the ENABLE statement is the **ENABLE block**. Each block following a HANDLE statement is called a **HANDLE block**.

A condition may be signaled explicitly by a SIGNAL statement.

R829	*signal-stmt*	**is** SIGNAL (*condition-name*) **or** SIGNAL (*)

Constraint: SIGNAL (*) is permitted only in a HANDLE block.

The blocks in an ENABLE construct may contain SIGNAL statements.

8.1.5.2 Execution of an ENABLE Construct. Upon execution of an ENABLE construct, first the specified conditions are enabled and then the ENABLE block is executed. If none of the enabled conditions is signaled, either by the processor or explicitly by a SIGNAL statement, the execution of the construct is complete when the execution of the block is complete.

8.1.5.2.1 Condition Enabling. Conditions are explicitly enabled by an ENABLE statement. Intrinsic conditions (8.1.5.4) also may be enabled by the processor.

All conditions that are enabled prior to an ENABLE statement remain enabled throughout the ENABLE construct. Conditions in the condition name list of the ENABLE statement are enabled only within the ENABLE construct.

8.1.5.2.2 Condition Signaling. Conditions may be signaled implicitly by the processor or explicitly by a SIGNAL statement.

The intrinsic conditions, if they are enabled, are signaled by the processor whenever the events they represent occur.

A condition is signaled **indeterminately** if it is detected during expression evaluation or assignment. An indeterminately signaled condition affects entities in the innermost ENABLE construct or program unit that contains the operation causing the signal (8.1.5.2). If circumstances are such that two independent operations could each signal a condition indeterminately in the same block, the condition that serves as the basis for transfer of control is processor dependent.

A condition is signaled **determinately** if it is detected in any other way. A determinately signaled condition can affect only entities in the statement in which the condition is detected (8.1.5.3).

Execution of a SIGNAL statement signals determinately the condition indicated by the condition name that appears in the statement. If the SIGNAL statement appears in a HANDLE block and the condition name is specified by *, the condition signaled is the condition that caused the transfer to the block. Signaling a dummy condition is equivalent to signaling the corresponding actual argument. A condition need not be enabled to be signaled explicitly.

8.1.5.2.3 Condition Handling. If a condition is signaled in an ENABLE block and the ENABLE construct contains a HANDLE block for that condition, the ENABLE construct is said to supply the handler for that condition. An ENABLE construct never supplies a handler for a condition detected in one of its handlers. The block following a HANDLE (*) statement is the default handler for that ENABLE construct. It handles for all conditions not otherwise handled explicitly in that ENABLE construct.

When a condition is signaled, control is transferred to the HANDLE block, supplied by the ENABLE construct in which the condition was enabled.

Execution of the HANDLE block completes the execution of the ENABLE construct.

If a condition is signaled but no handler is supplied, and the program unit is not a main program, the condition is propagated. If the current program unit is a function or subroutine, the condition is signaled in the invoking program unit. If the current program unit is a function or assignment subroutine (12.5.2.3), the condition is signaled indeterminately in the innermost ENABLE block or program unit invoking the current program unit. If the current program unit is a subroutine other than an assignment subroutine, the condition is signaled determinately in the statement invoking the current program unit.

If no handler exists for a signaled condition, program execution is terminated.

8.1.5.3 Effects of Signaling on Definition. The signaling of a condition may cause entities to become undefined. When a condition is signaled determinately in a statement, the entities affected are those whose definition status could have been affected by the statement had no condition been signaled. When a condition is signaled indeterminately in a block, the entities affected are those whose definition status has been affected or could have been affected by statements in the block had no condition been signaled.

8.1.5.4 Intrinsic Conditions. A processor must be able to detect the following list of **intrinsic conditions**:

(1) NUMERIC_ERROR. This condition occurs when the processor is unable to produce an acceptable result for an intrinsic numeric operation, either because the result is mathematically undefined or because the processor has no adequate representation for the mathematical result.

(2) BOUND_ERROR. This condition occurs when an array subscript, array section subscript, or substring range expression violates its bounds. Note that this does not include violations of the requirements derived from the size of an assumed-size array.

(3) IO_ERROR. This condition occurs when an error occurs in an input/output statement containing no IOSTAT= or ERR= specifier.

(4) END_OF_FILE. This condition occurs when an end-of-file condition (9.4.2.2) is encountered in an input statement containing no IOSTAT= or END= specifier.

(5) ALLOCATION_ERROR. This condition occurs when the processor is unable to perform an allocation requested by an ALLOCATE statement (6.2.1).

8.1.5.5 Examples of ENABLE Constructs. Example 1:

```
IO_CHECK:  ENABLE (IO_ERROR, END_OF_FILE)
   . . .
   READ (*, '(I5)') I
   . . .
   HANDLE (END_OF_FILE)
      I = -1
   HANDLE (IO_ERROR)
      I = 0
   END ENABLE IO_CHECK
   . . .
```

Example 2:

```
ENABLE (SINGULARITY_ERROR)
   . . .
   DO (5 TIMES)
      CALL MAT_INV (MATRIX, VMATRIX, SDET, DONE)
      IF (DONE) EXIT
      ! RESCALE PROBLEM AND TRY AGAIN
   END DO
   IF (.NOT. DONE) SIGNAL (SINGULARITY_ERROR)
   . . .
```

```
HANDLE (SINGULARITY_ERROR)
   WRITE (3, *) "CANNOT INVERT MATRIX"
   STOP
END ENABLE

CONTAINS
REAL FUNCTION DETERMINANT (X, N)
   REAL X (N, N)
   . . .
   IF (DIAG == 0.0) SIGNAL (SINGULARITY_ERROR)
   . . .
END FUNCTION DETERMINANT

SUBROUTINE MAT_INV (MAT, INV, DET)
   FLAG = .TRUE.
   . . .
   ENABLE (NUMERIC_ERROR)
   . . .
      DET = DETERMINANT (MAT, N)
   . . .
   HANDLE (NUMERIC_ERROR)
   . . .
   HANDLE (SINGULARITY_ERROR)
      INV = 0.0
      DET = 0.0
      FLAG = .FALSE.
   END ENABLE
END SUBROUTINE MAT_INV
END
```

When SINGULARITY_ERROR is signaled in DETERMINANT, the signal is propagated to MAT_INV, which contains the handler for that condition. When the same condition is signaled in the main program, the handler in the main program is executed.

8.2 Branching. **Branching** is used to alter the normal execution sequence. A branch causes a transfer of control from one statement in a program unit to a labeled branch target statement in the same program unit. A **branch target statement** is an *action-stmt*, an *end-program-stmt*, an *end-function-stmt*, an *end-subroutine-stmt*, an *if-then-stmt*, an *end-if-stmt*, a *select-stmt*, an *end-select-stmt*, a *do-stmt*, a *do-termination*, an *enable-stmt*, an *end-enable-stmt*, or a *where-construct-stmt*.

It is permissible to branch to an END SELECT statement only from within its CASE construct.

It is permissible to branch to a DO termination only from within its DO construct.

It is permissible to branch to an END ENABLE statement only from within its ENABLE construct.

8.2.1 Statement Labels. **Statement labels** provide a means of referring to individual statements. Any statement may be identified with a label, but only branch target statements, FORMAT statements, and DO terminations may be referred to by the use of statement labels (3.3.3).

8.2.2 GO TO Statement.

R830 *goto-stmt* **is** GO TO *label*

Constraint: *label* must be the statement label of a *branch-target* that appears in the same program unit as the *go-to-stmt*.

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

8.2.3 Computed GO TO Statement.

R831 *computed-goto-stmt* is GO TO (*label-list*) [,] *scalar-int-expr*

Constraint: Each *label* in *label-list* must be the statement label of a branch target that appears in the same program unit as the *computed-goto-stmt*.

The same statement label may appear more than once in a label list.

Execution of a computed GO TO statement causes evaluation of the scalar integer expression. If this value is *i* such that $1 \le i \le n$ where *n* is the number of labels in *label-list*, a transfer of control occurs so that the next statement executed is the one identified by the *i*th label in the list of labels. If *i* is less than 1 or greater than *n*, the execution sequence continues as though a CONTINUE statement were executed.

8.2.4 ASSIGN and Assigned GO TO Statement.

R832 *assign-stmt* is ASSIGN *label* TO *scalar-int-variable*

Constraint: *label* must be the statement label of a branch target or a *format-stmt*.

R833 *assigned-goto-stmt* is GO TO *scalar-int-variable* [[,] (*label-list*)]

Constraint: Each *label* in *label-list* must be the statement label of a branch target that appears in the same program unit as the *assigned-goto-stmt*.

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

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

When an assigned GO TO statement is executed, its integer variable must be defined with the label of a branch target. When an input/output statement containing the integer variable as a format identifier (9.4.1.1) is executed, the integer variable must be defined with the label of a FORMAT statement. While defined with a statement label value, the integer variable must not be referenced in any other context.

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

At the time of execution of an assigned GO TO statement, the integer variable must be defined with the value of a statement label of a branch target that appears in the same program unit. Note that the variable may be defined with a statement label value only by an ASSIGN statement 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 branch target identified by the statement label currently assigned to the integer variable is executed next.

If the parenthesized list is present, the statement label assigned to the integer variable must be one of the statement labels in the list.

8.2.5 Arithmetic IF Statement.

R834 *arithmetic-if-stmt* is IF (*scalar-numeric-expr*) *label, label, label*

Constraint: Each *label* must be the label of a branch target that appears in the same program unit as the *arithmetic-if-stmt*.

Constraint: The *scalar-numeric-expr* must not be a complex expression.

The same label may appear more than once in one arithmetic IF statement.

Execution of an arithmetic IF statement causes evaluation of the numeric expression followed by a transfer of control. The branch target identified by the first label, the second label, or the third label is executed next as the value of the numeric expression is less than zero, equal to zero, or greater than zero, respectively.

8.3 CONTINUE Statement.

Execution of a CONTINUE statement has no effect.

R835 *continue-stmt* **is** CONTINUE

CONTINUE statements are usually identified by labels that also appear in control statements, such as the DO statement.

8.4 STOP Statement.

R836 *stop-stmt* **is** STOP [*access-code*]

R837 *access-code* **is** *char-constant*
or *digit* [*digit* [*digit* [*digit* [*digit*]]]]

Execution of a STOP statement causes termination of execution of the executable program. At the time of termination, the access code if any, is accessible. Leading zero digits are significant.

8.5 PAUSE Statement.

R838 *pause-stmt* is PAUSE [*access-code*]

Execution of a PAUSE statement causes a suspension of execution of the executable program. Execution must be resumable. At the time of suspension of execution, the access code is accessible. 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. Leading zero digits in the access code are significant.

9 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 describe or inquire about the properties of the connection to the external medium.

The input/output statements are the OPEN, CLOSE, READ, WRITE, PRINT, BACKSPACE, ENDFILE, REWIND, and INQUIRE statements.

The READ statement is a **data transfer input statement**. The WRITE statement and the PRINT statement are **data transfer output statements**. The OPEN statement and the CLOSE statement are **file connection statements**. The INQUIRE statement is a **file inquiry statement**. The BACKSPACE, ENDFILE, and REWIND statements are **file positioning statements**.

9.1 Records.

9.1 Records. A **record** is a sequence of values or a sequence of characters. For example, a line on a terminal 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

9.1.1 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.

Formatted records may be prepared by means other than Fortran; for example, by some manual input device.

9.1.2 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 input/output list (9.4.2) used when it is written, as well as on the processor and the external medium. The length may be zero.

Unformatted records may be read or written only by unformatted input/output statements.

9.1.3 Endfile Record. An **endfile record** is written explicitly by the ENDFILE statement. The file must be connected for sequential access. An endfile record is written implicitly to a file connected for sequential access when the last operation on the file is an output statement other than the ENDFILE statement, and:

(1) A REWIND or BACKSPACE statement references the unit, or

(2) The unit (file) is closed, either explicitly by a CLOSE statement or implicitly by a program termination not caused by an error condition.

An endfile record may occur only as the last record of a file. An endfile record does not have a length property.

9.2 Files.

9.2 Files. A **file** is a sequence of records.

There are two kinds of files:

(1) External

(2) Internal

9.2.1 External Files. An **external file** is any file that exists in a medium external to the executable program.

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. The set of allowable names is processor dependent and may be empty.

An external file that is connected to a unit has a **position** property (9.2.1.3).

9.2.1.1 File Existence. At any given time, there is a processor-determined set of external 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, there may be security reasons that 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. An INQUIRE, OPEN, CLOSE, WRITE, PRINT, REWIND, or ENDFILE statement may also refer to a file that does not exist.

9.2.1.2 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 (9.3.2).

9.2.1.2.1 Sequential Access. When connected for sequential access, an external 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. In this case, 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 must be an endfile record.

(3) The records of the file must not be read or written by direct access input/output statements.

9.2.1.2.2 Direct Access. When connected for direct access, an external file has the following properties:

(1) 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. 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.

(4) All records of the file have the same length.

(5) Records need not be read or written in the order of their record numbers. Any record may be written into the file while it is connected 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 has been written since the file was created.

(6) The records of the file must not be read or written using list-directed (10.8) or name-directed formatting (10.9).

9.2.1.3 File Position. 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 $1 < i \leq n$ and a file is positioned within the ith record or between the $(i - 1)$th record and the ith record, the $(i - 1)$th record is the **preceding record.** If $n \geq 1$ and the file is positioned at its terminal point, the preceding record is the nth 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 $1 \leq i < n$ and a file is positioned within the ith record or between the ith and $(i + 1)$th record, the $(i + 1)$th record is the **next record.** If $n \geq 1$ 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 nth (last) record, there is no next record.

9.2.1.3.1 File Position Prior to Data Transfer. The positioning of the file prior to data transfer depends on the method of access: sequential or direct.

For 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.

For direct access, the file is positioned at the beginning of the record specified by the record specifier. This record becomes the current record.

If the file contains an endfile record, the file must not be positioned after the endfile record prior to data transfer.

9.2.1.3.2 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 REWIND or BACKSPACE statement may be used to reposition the file.

If an error condition exists, the position of the file is indeterminate.

9.2.2 Internal Files. Internal files provide a means of transferring and converting data from internal storage to internal storage.

9.2.2.1 Internal File Properties. An internal file has the following properties:

(1) The file is a character variable other than an array section with any vector subscripts.

(2) A record of an internal file is a scalar character variable.

(3) If the file is a scalar character variable, it consists of a single record whose length is the same as the length of the scalar character variable. If the file is a character array or array section, it is treated as a sequence of character array elements. Each array element, if any, 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 (6.2.4.2) or the array section (6.2.4.3). Every record of the file has the same length, which is the length of an array element in the array.

(4) A 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. If the number of characters to be written is greater than the length of the record, the effect is as though characters equal to the length are written and remaining characters truncated.

(5) A record may be read only if the record is defined.

(6) A record of an internal file may become defined (or undefined) by means other than an output statement. For example, the character variable 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.

9.2.2.2 Internal File Restrictions. An internal file has the following restrictions:

(1) Reading and writing records must be accomplished only by sequential access formatted input/output statements that do not specify name-directed formatting.

(2) An internal file must not be specified in a file connection statement, a file positioning statement, or a file inquiry statement.

9.3 File Connection.

A **unit**, specified by an *io-unit*, provides a means for referring to a file.

R901	*io-unit*	**is** *external-file-unit*
		or *
		or *internal-file-unit*
R902	*external-file-unit*	**is** *scalar-int-expr*
R903	*internal-file-unit*	**is** *char-variable*

A *scalar-int-expr* that identifies an external file unit must be zero or positive.

The *io-unit* in a file positioning statement, a file connection statement, or a file inquiry statement must not be an asterisk or an *internal-file-unit*.

The external unit identified by the value of *scalar-int-expr* 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 asterisk identifies a particular processor-dependent external unit that is preconnected for formatted sequential access.

9.3.1 Unit Existence. At any given time, there is a processor-determined set of units that are said to exist for an executable program.

All input/output statements may refer to units that exist. The INQUIRE statement and the CLOSE statement also may refer to units that do not exist.

9.3.2 Connection of a File to 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 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.

All input/output statements except an OPEN, a CLOSE, or an INQUIRE statement 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.

A unit must not be connected to more than one file at the same time, and a file must not be connected to more than one unit at the same time. However, 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 to 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 to a different unit. Note, however, that the only means of referencing a file that has been disconnected is by the appearance of its name in an OPEN or INQUIRE statement. There may be no means of reconnecting an unnamed file once it is disconnected.

9.3.3 Preconnection. Preconnection means that the unit is connected to a file at the beginning of execution of the executable program and therefore it may be specified in input/output statements without the prior execution of an OPEN statement.

9.3.4 The OPEN Statement. The **OPEN statement** may be used to connect an existing file to a unit, create a file 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.

An external unit may be connected by an OPEN statement in any program unit of an executable program and, once connected, a reference to it may appear in any program unit of the executable program.

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 an 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 an 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 without a STATUS= specifier had been executed for the unit immediately prior to the execution of an OPEN statement.

If the file to be connected to the unit is the same as the file to which the unit is already connected, only the BLANK=, DELIM=, PAD=, ERR=, and IOSTAT= specifiers may have a value different from the one currently in effect. Execution of an OPEN statement causes the new value of the BLANK=, DELIM=, and PAD= specifiers to be in effect. The position of the file is unaffected. The values of specifiers other than BLANK= , DELIM= , PAD= , ERR= , and IOSTAT= remain in effect.

If a file is already connected to a unit, execution of an OPEN statement on that file and a different unit is not permitted.

R904	*open-stmt*	**is** OPEN (*connect-spec-list*)
R905	*connect-spec*	**is** [UNIT=] *external-file-unit*
		or IOSTAT= *iostat-variable*
		or ERR= *label*
		or FILE= *scalar-char-expr*
		or STATUS= *scalar-char-expr*
		or ACCESS= *scalar-char-expr*
		or FORM= *scalar-char-expr*
		or RECL= *scalar-int-expr*
		or BLANK= *scalar-char-expr*
		or POSITION= *scalar-char-expr*
		or ACTION= *scalar-char-expr*
		or DELIM= *scalar-char-expr*
		or PAD= *scalar-char-expr*

Constraint: Each specifier must not appear more than once in a given *open-stmt*; the UNIT= specifier must appear.

Constraint: If the STATUS= specifier is 'OLD' or 'NEW', the FILE= specifier must be present.

Constraint: If the STATUS= specifier is 'SCRATCH', the FILE= specifier must be absent.

Constraint: The RECL= specifier must be given when the ACCESS= specifier evaluates to DIRECT; otherwise, it must be omitted.

A specifier that requires a *scalar-char-expr* may have a limited list of character values. These values are listed for each such specifier. Any trailing blanks are ignored. If a processor is capable of representing letters in both upper and lower case, the value specified is as if all letters were in upper case. Some specifiers have an assumed value if the specifier is omitted.

The IOSTAT= specifier and ERR= specifier are described in Sections 9.4.1.4 and 9.4.1.5, respectively.

9.3.4.1 FILE= Specifier in the OPEN Statement. The value of the FILE= specifier is the name of the file to be connected to the specified unit. 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 may become connected to a processor-determined file.

9.3.4.2 STATUS= Specifier in the OPEN Statement. The *scalar-char-expr* must evaluate to 'OLD', 'NEW', 'SCRATCH', or 'UNKNOWN'. 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. 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 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.

9.3.4.3 ACCESS= Specifier in the OPEN Statement. The *scalar-char-expr* must evaluate to 'SEQUENTIAL' or 'DIRECT'. The ACCESS= specifier specifies the access method for the connection of the file as being sequential or direct. 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. For a new file, the processor creates the file with a set of allowed access methods that includes the specified method.

9.3.4.4 FORM= Specifier in the OPEN Statement. The *scalar-char-expr* must evaluate to 'FORMATTED' or 'UNFORMATTED'. The FORM= specifier determines whether the file is being connected for formatted or unformatted input/output. If the FORM= 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. For a new file, the processor creates the file with a set of allowed forms that includes the specified form.

9.3.4.5 RECL= Specifier in the OPEN Statement. The value of the RECL= specifier must be positive.

It specifies the length of each record in a file being connected for direct 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 input/output, the length is measured in processor-dependent units. For an existing file, the value of the RECL= specifier must be included in the set of allowed record lengths for the file. For a new file, the processor creates the file with a set of allowed record lengths that includes the specified value.

9.3.4.6 BLANK= Specifier in the OPEN Statement. The *scalar-char-expr* must evaluate to 'NULL' or 'ZERO'. The BLANK= specifier is permitted only for a file being connected for formatted input/output. 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 the BLANK= specifier is omitted, a value of NULL is assumed.

9.3.4.7 POSITION= Specifier in the OPEN Statement. The *scalar-char-expr* must evaluate to 'ASIS', 'REWIND', or 'APPEND'. ASIS causes the file to be opened without changing its position. REWIND positions the file at its initial point. APPEND positions a file that exists at its terminal point such that the endfile record is the next record. A file that does not exist (a NEW file, either specified explicitly or by default) is positioned at its initial point. The connection must be for sequential access. If this specifier is omitted, the value ASIS is assumed.

9.3.4.8 ACTION= Specifier in the OPEN Statement. The *scalar-char-expr* must evaluate to 'READ', 'WRITE', or 'READ/WRITE'. READ specifies that the WRITE, PRINT, and ENDFILE statements may not refer to this connection. WRITE specifies that READ statements may not refer to this connection, READ/WRITE permits any I/O statements to refer to this connection. If this specifier is omitted, the value READ/WRITE is assumed.

9.3.4.9 DELIM= Specifier in the OPEN Statement. The *scalar-char-expr* must evaluate to 'APOSTROPHE', 'QUOTE', or 'NONE'. If APOSTROPHE is specified, the apostrophe will be used to delimit character constants written with list-directed or name-directed formatting and all internal apostrophes will be doubled. If QUOTE is specified, the quotation mark will be used to delimit character constants written with list-directed or name-directed formatting and all internal quotation marks will be doubled. If the value of this specifier is NONE, the character constant when written will not be delimited by apostrophes or quotation marks. If this specifier is omitted, a value of NONE is assumed. This specifier is permitted only for a file being connected for formatted input/output. This specifier is ignored during input of a formatted record.

9.3.4.10 PAD= Specifier in the OPEN Statement. The *scalar-char-expr* must evaluate to 'YES' or 'NO'. If YES is specified, a formatted input record is logically padded with blanks when an input list is specified and the format specification requires more data from a record than the record contains. If NO is specified, the input list and the format specification must not require more characters from a record than the record contains. If this specifier is omitted, a value of NO is assumed.

9.3.5 The CLOSE Statement. The **CLOSE statement** is used to terminate the connection of a particular file to a unit.

R906	*close-stmt*	**is** CLOSE (*close-spec-list*)
R907	*close-spec*	**is** [UNIT=] *external-file-unit*
		or IOSTAT= *iostat-variable*
		or ERR= *label*
		or STATUS= *scalar-char-expr*

Constraint: A given specifier must not appear more than once in a given *close-stmt*; the unit specifier must appear.

Constraint: The *io-unit* must be an *external-file-unit*.

The IOSTAT= specifier and ERR= specifier are described in Sections 9.4.1.4 and 9.4.1.5, respectively.

A specifier that requires a *scalar-char-expr* may have a limited list of character values. These values are listed for each such specifier. Any trailing blanks are ignored. If a processor is capable of representing letters in both upper and lower case, the value specified is as if all letters were in upper case. Some specifiers have an assumed value if the specifier is omitted.

9.3.5.1 STATUS= Specifier in the CLOSE Statement. The *scalar-char-expr* must evaluate to 'KEEP' or 'DELETE'. The STATUS= specifier 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 a CLOSE statement is SCRATCH. If KEEP is specified for a file that exists, the file continues to exist after the execution of a CLOSE statement. If KEEP is specified for a file that does not exist, the file will not exist after the execution of a CLOSE statement. If DELETE is specified, the file will not exist after the execution of a 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.

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.

9.4 Data Transfer Statements. The **READ statement** is the data transfer input statement. The **WRITE statement** and **PRINT statement** are the data transfer output statements.

Termination of an input/output data transfer statement occurs when any of the following conditions are met:

(1) All elements of the *input-item-list* or *output-item-list* have been read or written, with or without editing, to or from the specified file.

(2) An error condition is encountered.

(3) An end-of-file condition is encountered.

(4) An end-of-record mark (/) is encountered in the record being read during list-directed or name-directed input.

R908	*read-stmt*	**is** READ (*io-control-spec-list*) [*input-item-list*]
		or READ *format* [, *input-item-list*]
R909	*write-stmt*	**is** WRITE (*io-control-spec-list*) [*output-item-list*]
R910	*print-stmt*	**is** PRINT *format* [, *output-item-list*]

9.4.1 Control Information List. The *io-control-spec-list* is a **control information list** that includes:

(1) A reference to the source or destination of the data to be transferred

(2) Optional specification of editing processes

(3) Optional specification to identify a record

(4) Optional specification of exception handling

(5) Optional return of counts of values transmitted and values skipped

(6) Optional return of status

The control information list governs the data transfer.

R911	*io-control-spec*	**is** [UNIT=] *io-unit*
		or [FMT=] *format*
		or REC= *scalar-int-expr*
		or PROMPT= *scalar-char-expr*
		or IOSTAT= *iostat-variable*
		or ERR= *label*
		or END= *label*
		or NULLS= *scalar-int-variable*
		or VALUES= *scalar-int-variable*

Constraint: An *io-control-spec-list* must contain exactly one *io-unit* and may contain at most one of each of the other specifiers.

Constraint: A NULLS= specifier must not appear in a *write-stmt* or *print-stmt*.

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=.

If the unit specifier specifies an internal file, the *io-control-spec-list* must contain a *format* other than ** and must not contain a REC= specifier.

9.4.1.1 Format Specifier.

R912	*format*	**is** *char-expr*
		or *label*
		or *
		or **
		or *scalar-int-variable*

The *label* must be the statement label of a FORMAT statement.

The *scalar-int-variable* must have been assigned (8.2.4) the statement label of a FORMAT statement that appears in the same program unit as the *format*.

The *scalar-char-expr* must evaluate to a character object that is a valid format item list (10.2). Note that *scalar-char-expr* includes a character constant.

If the control information list contains a *format*, the statement is a **formatted input/output statement**; otherwise, it is an **unformatted input/output statement**.

If *format* is *, the statement is a **list-directed input/output statement**. If *format* is **, the statement is a **name-directed input/output statement**. A REC= specifier must not be present when *format* is * or **.

9.4.1.2 Record Number. The REC= specifier specifies the number of the record that is to be read or written in a file connected for direct access. If the control information list contains a REC= specifier, the statement is a **direct access input/output statement** an END= specifier must not be present; otherwise, it is a **sequential access input/output statement**.

9.4.1.3 Prompt Specifier. For a formatted external READ statement, the *scalar-char-variable* specified in the PROMPT= specifier is written to the connected unit without line spacing following it. The input statement is then executed. If the connection is to a device that does not permit both input and output, the PROMPT= specifier is ignored. The PROMPT+ specifier is not permitted in a WRITE statement.

9.4.1.4 Input/Output Status.

R913 *iostat-variable* **is** *scalar-int-variable*

Execution of an input/output statement containing the IOSTAT= specifier causes *iostat-variable* 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. Note that this condition may occur only during a sequential input statement.

Consider the example:

```
READ (FMT = "(E8.3)", UNIT=3, IOSTAT = IOSS) X

IF (IOSS < 0) THEN
   !
   ! PERFORM END-OF-FILE PROCESSING ON THE FILE
   ! CONNECTED TO UNIT 3.

   CALL END_PROCESSING

ELSE IF (IOSS > 0) THEN

   ! PERFORM ERROR PROCESSING

   CALL ERROR_PROCESSING

END IF
```

9.4.1.5 Error Branch. If an input/output statement contains an ERR= 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 statement becomes indeterminate,

(3) If the input/output statement also contains an *iostat-variable*, the *iostat-variable* becomes defined with a processor-dependent positive integer value, and

(4) Execution continues with the statement specified in the ERR= specifier. The statement label must be in the same program unit as the input/output statement.

The statement label must be in the same program unit as the input/output statement.

9.4.1.6 End of File Branch. If an input statement contains an END= specifier and the processor encounters an end-of-file condition and encounters no error condition during execution of the statement:

(1) Execution of the READ statement terminates,

(2) If the input statement also contains an IOSTAT= specifier, the *iostat-variable* becomes defined with a processor-dependent negative integer value, and

(3) Execution continues with the statement specified in the END= specifier. The statement label must be in the same program unit as the input/output statement.

In a WRITE statement, the control information list must not contain an END= specifier.

The statement label must be in the same program unit as the input/output statement.

9.4.1.7 Nulls Count. A **null value** is a value that 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 must 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.

When an input statement terminates, the variable specified in the NULLS= specifier is defined to be the count of the null values read by the input statement. The value of the variable can be nonzero only for list-directed or name-directed input.

9.4.1.8 Values Count. When an input/output statement terminates, the variable specified in the VALUES= specifier is defined to be the count of the number of values successfully read or written, with or without editing, by the input/output statement.

Any null values are included in the count of values.

9.4.2 Data Transfer Input/Output List. An input/output list specifies the entities whose values are transferred by a data transfer input/output statement.

R914	*input-item*	**is** *variable*
		or *io-implied-do*
R915	*output-item*	**is** *expr*
		or *io-implied-do*
R916	*io-implied-do*	**is** (*io-implied-do-object-list* , *io-implied-do-control*)
R917	*io-implied-do-object*	**is** *input-item*
		or *output-item*
R918	*io-implied-do-control*	**is** *scalar-numeric-expr* , ☐

□ *scalar-numeric-expr* , [*scalar-numeric-expr*]

Constraint: In an *input-item-list*, an *io-implied-do-object* must be an *input-item*. In an *output-item-list*, an *io-implied-do-object* must be an *output-item*.

Constraint: An *input-item* must not appear as, nor be associated with, the *do-variable* of any *io-implied-do* that contains the *input-item*.

Constraint: The *do-variable* of an *io-implied-do* that is contained within another *io-implied-do* must not appear as, nor be associated with, the *do-variable* of the containing *io-implied-do*.

If an array name or array section name appears as an input/output list item, it is treated as if all of the elements, if any, of the array or array section were specified in the order given by the ordering of the array elements in the array (6.2.4.2) or the array section (6.2.4.3). The name of an assumed-size dummy array must not appear as an input/output list item. An input/output list for a name-directed formatted data transfer input/output statement must not contain an *io-implied-do*.

If the name of a derived-type object appears as an input/output list item, it is treated as if all of the components of the object were specified in the same order as in the definition of the derived type. Note that in the case of an input list item of a derived type having a variant component, the selection of the set of components to correspond to the variant component (4.4.1.2) is affected by the transmission of a value (other than a null value) to the tag component. (See 9.4.3.4.)

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.

An *io-implied-do* must not appear in the input/output list of a name-directed formatted data transfer input/output statement.

9.4.2.1 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 connected for sequential access. 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.

Note that an end-of-file condition can occur at the beginning of an input statement or within a formatted input statement when more than one record is required by the interaction of the input/output list and the format.

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. The VALUES= specifier, if any, is defined with the count of values successfully read or written. Any remaining list items of the input/output list are undefined. For any specific error condition, the number of values defined is processor dependent. Note that for list-directed and name-directed input, some elements of the input/output list may not have had their definition status changed due to null values.

Let *n* be the value of the variable specified in a VALUES= specifier. If the *n*th value of an input/output list, when related to the format list by the normal matching process, is in the range of one or more *io-implied-dos*, the DO variable is defined with the count of values successfully transferred for that *io-implied-do*. Any DO variable defined prior to the occurrence of the error condition in the matching process remains defined. Any remaining *do-variable* in the input/output list are undefined.

If an error condition occurs during execution of an input/output statement that contains neither an IOSTAT= nor an ERR= specifier, or if an end-of-file condition occurs during execution of a READ statement that contains neither an *iostat-variable* nor an END= specifier, the intrinsic condition IO_ERROR is signaled.

9.4.3 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 one 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)

(6) Position the file after data transfer

(7) Cause *iostat-variable* (if any) to become defined, and cause the variables in the VALUES= and NULLS= specifiers, if specified, to become defined.

9.4.3.1 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 or PRINT statement causes values to be transferred to a file from the entities specified by the output list and format specification, if any. Execution of a WRITE or PRINT statement for a file that does not exist creates the file unless an error condition occurs.

9.4.3.2 Identifying a Unit. A data transfer input/output statement that contains an input/output control list includes a file unit specifier that identifies an external unit or an internal file. A READ statement that does not contain an input/output control list specifies a particular processor-determined unit, which is the same as the unit identified by * in a READ statement that contains an input/output control list. The PRINT statement specifies some other processor-determined unit, which is the same as the unit identified by * in a WRITE statement. Thus, each data transfer input/output statement identifies an external unit or an internal file.

The unit identified by a data transfer input/output statement must be connected to a file when execution of the statement begins. Note that the file may be preconnected.

9.4.3.3 Establishing a Format. If the input/output control list contains * as a format, list-directed formatting is established. If the format is **, name-directed formatting is established.

Otherwise, the format specification identified by the format specifier is established. If the format is an array, the effect is as if all elements of the array were concatenated in subscript order value.

On output, if an internal file has been specified, a format specification that is in the file or is associated with the file must not be specified.

9.4.3.4 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 for all data transfer input/output statements except name-directed formatted data transfer input statements. The list items for a name-directed formatted data transfer input statement are processed in the order of the entities specified within the input records.

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 for all data transfer input/output statements except name-directed formatted data transfer input statements. 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.

All values following the *name*= part of the name-directed entity (10.9) within the input records are transmitted to the matching entity specified by the list item prior to processing any succeeding entity within the input record for name-directed formatted data transfer input statements. If an entity is specified more than once within the input record during a name-directed formatted data transfer input statement, the last occurrence of the entity specifies the value or values to be used for that entity.

An input list item, or an entity associated with it, 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 *io-implied-do*.

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.

9.4.3.4.1 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.

On output to a file connected for direct access, the output list must not specify more values than have been specified by the RECL= specifier.

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.

9.4.3.4.2 Formatted Data Transfer. During formatted data transfer, data are transferred with editing between the entities specified by the input/output list and the file. Format control is initiated and editing is performed as described in Section 10. The current record and possibly additional records are read or written.

Objects of intrinsic or derived types may be transferred through a formatted data transfer statement. However, the requirement that the format be established prior to any transfer of data (9.4.3) and the possibility of variant components may effectively prevent explicitly formatted (10.1) input to objects of derived types containing variant components, because of the interdependence of the input/output list and format specification.

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.

On input, the input list and format specification must not require more characters from a record than the record contains. However, blank padding to satisfy this condition may be specified by a PAD= specifier in an OPEN statement.

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 have been specified by the RECL= specifier.

9.4.3.5 List-Directed Formatting. If list-directed formatting has been established, editing is performed as described in Section 10.8.

9.4.3.6 Name-Directed Formatting. The characters in one or more name-directed records constitute a sequence of name, values, and value separators.

If name-directed formatting has been established, editing is performed as described in Section 10.9.

9.4.4 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, the vertical spacing is one line and no characters other than blank are printed in that line.

The PRINT statement does not imply that printing will occur, and the WRITE statement does not imply that printing will not occur.

9.5 File Positioning Statements.

R919 *backspace-stmt* **is** BACKSPACE *external-file-unit*
or BACKSPACE (*position-spec-list*)

R920 *endfile-stmt* **is** ENDFILE *external-file-unit*
or ENDFILE (*position-spec-list*)

R921 *rewind-stmt* **is** REWIND *external-file-unit*
or REWIND (*position-spec-list*)

Constraint: BACKSPACE, ENDFILE, and REWIND apply only to files connected for sequential access.

R922 *position-spec* **is** [UNIT =] *external-file-unit*
or IOSTAT = *iostat-variable*
or ERR = *label*

Constraint: A *position-spec-list* must contain exactly one *external-file-unit* and may contain at most one of each of the other specifiers.

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 or name-directed formatting is prohibited.

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.

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.

9.6 File Inquiry. The 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 performed according to the rules for assignment statements.

An INQUIRE statement may be executed before, while, or after a file is connected to a unit. All values assigned by an INQUIRE statement are those that are current at the time the statement is executed.

R923	*inquire-stmt*	**is** INQUIRE (*inquire-spec-list*) [*output-item-list*]

9.6.1 Inquiry Specifiers. Unless constrained, the following inquiry specifiers may be used in either form of the INQUIRE statement:

R924	*inquire-spec*	**is** FILE = *scalar-char-expr*
		or UNIT = *external-file-unit*
		or IOSTAT = *iostat-variable*
		or ERR = *label*
		or EXIST = *scalar-logical-variable*
		or OPENED = *scalar-logical-variable*
		or NUMBER = *scalar-int-variable*
		or NAMED = *scalar-logical-variable*
		or NAME = *scalar-char-variable*
		or ACCESS = *scalar-char-variable*
		or SEQUENTIAL = *scalar-char-variable*
		or DIRECT = *scalar-char-variable*
		or FORM = *scalar-char-variable*
		or FORMATTED = *scalar-char-variable*
		or UNFORMATTED = *scalar-char-variable*
		or RECL = *scalar-int-variable*
		or NEXTREC = *scalar-int-variable*
		or BLANK = *scalar-char-variable*
		or POSITION = *scalar-char-variable*
		or ACTION = *scalar-char-variable*
		or DELIM = *scalar-char-variable*
		or PAD = *scalar-char-variable*
		or IOLENGTH = *scalar-int-variable*

Constraint: An INQUIRE statement must contain one FILE= specifier or one UNIT= specifier, but not both, and at most one of each of the other specifiers.

Constraint: The IOLENGTH= specifier and the *output-item-list* must both appear if either appears.

When a returned value is of type character and the processor is capable of representing letters in both upper and lower case, the value returned is in upper case.

If an error condition occurs during execution of an INQUIRE statement, all of the inquiry specifier variables except the one in the IOSTAT= specifier become undefined.

Note that the specifier variables in the EXIST= and OPENED= specifiers always become defined unless an error condition occurs.

9.6.1.1 FILE= Specifier in the INQUIRE Statement. The value of *scalar-char-expr* in the FILE= specifier 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 *scalar-char-expr* must be of a form acceptable to the processor as a file name.

9.6.1.2 EXIST= Specifier in the INQUIRE Statement. Execution of an INQUIRE by file statement causes *scalar-logical-variable* in the EXIST= specifier to be assigned the value true if there exists a file with the specified name; otherwise, false is assigned. Execution of an INQUIRE statement by unit causes true to be assigned if the specified unit exists; otherwise, false is assigned.

9.6.1.3 OPENED= Specifier in the INQUIRE Statement. Execution of an INQUIRE by file statement causes *scalar-logical-variable* in the OPENED= specifier to be assigned the value true if the file specified is connected to a unit; otherwise, false is assigned. Execution of an INQUIRE statement by unit causes *scalar-logical-variable* to be assigned the value true if the specified unit is connected to a file; otherwise, false is assigned.

9.6.1.4 NUMBER= Specifier in the INQUIRE Statement. The *scalar-int-variable* in the NUMBER= specifier is assigned the value of the external unit identifier of the unit that is currently connected to the file. If there is no unit connected to the file, the value −1 is assigned.

9.6.1.5 NAMED= Specifier in the INQUIRE Statement. The *scalar-logical-variable* in the NAMED= specifier is assigned the value true if the file has a name; otherwise, it is assigned the value false.

9.6.1.6 NAME= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the NAME= specifier is assigned the value of the name of the file if the file has a name; otherwise, it becomes undefined. 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 *scalar-char-expr* in the FILE= specifier in an OPEN statement.

9.6.1.7 ACCESS= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the ACCESS= specifier 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, it is assigned the value UNDEFINED.

9.6.1.8 SEQUENTIAL= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the SEQUENTIAL= specifier 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.

9.6.1.9 DIRECT= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the DIRECT= specifier 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.

9.6.1.10 FORM= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the FORM= specifier is assigned the value FORMATTED if the file is connected for formatted input/output, and is assigned the value UNFORMATTED if the file is connected for unformatted input/output. If there is no connection, it is assigned the value UNDEFINED.

9.6.1.11 FORMATTED= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the FORMATTED= specifier 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.

9.6.1.12 UNFORMATTED= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the UNFORMATTED= specifier 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.

9.6.1.13 RECL= Specifier in the INQUIRE Statement. The *scalar-int-variable* in the RECL= specifier 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 or if the connection is not for direct access, *scalar-int-variable* becomes undefined.

9.6.1.14 NEXTREC= Specifier in the INQUIRE Statement. The *scalar-int-variable* in the NEXTREC= specifier 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, *scalar-int-variable* 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, *scalar-int-variable* becomes undefined.

9.6.1.15 BLANK= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the BLANK= specifier 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, *scalar-char-variable* is assigned the value UNDEFINED.

9.6.1.16 POSITION= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the POSITION= specifier is assigned the value REWIND if the file is connected by an OPEN statement for positioning at its initial point, APPEND if the file is connected for positioning at its terminal point, and ASIS if the file is connected without changing its position. If there is no connection, *scalar-char-variable* is assigned the value UNDEFINED. If the file has been repositioned since the connection, *scalar-char-variable* is assigned the value UNDEFINED.

9.6.1.17 ACTION= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the ACTION= specifier is assigned the value READ if the file is connected for input only, WRITE if the file is connected for output only, and READ/WRITE if it is connected for both input and output. If there is no connection, *scalar-char-variable* is assigned the value UNDEFINED.

9.6.1.18 DELIM= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the DELIM= specifier is assigned the value APOSTROPHE if the apostrophe is to be used to delimit character data written by list-directed or name-directed formatting. If the quotation mark is used to delimit these data, the value QUOTE is assigned. If neither the apostrophe nor the quote is used to delimit the character data, the value NONE is assigned. If there is no connection or if the connection is not for formatted input/output, *scalar-char-variable* is assigned the value UNDEFINED.

9.6.1.19 PAD= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the PAD= specifier is assigned the value YES if the connection of the file to the unit included the PAD= specifier and its value was YES. Otherwise, *scalar-char-variable* is assigned the value NO.

9.6.1.20 IOLENGTH= Specifier in the INQUIRE Statement. The *scalar-int-variable* in the IOLENGTH= specifier is assigned the processor-dependent value that results from the use of the input/output list in an unformatted output statement. It must be suitable as a RECL= specifier in an OPEN statement that connects a file for unformatted direct access when there are input/output statements with the same input/output list.

9.6.1.21 Restrictions on Inquiry Specifiers. A variable that may become defined or undefined as a result of its use as a specifier in an INQUIRE statement, or any associated entity, must not appear as another specifier in the same INQUIRE statement.

The *inquire-spec-list* in an INQUIRE by file statement must contain exactly one FILE= specifier and must not contain a UNIT= specifier.

Execution of an INQUIRE by file statement causes the variables specified in the NAMED=, SEQUENTIAL=, DIRECT=, FORMATTED=, and UNFORMATTED= specifiers to be assigned values only if the *external-file-unit* specified is acceptable to the processor as a file name and if there exists a file with the name; otherwise, the *scalar-logical-variable* in the NAMED= specifier is assigned a value of false, the *scalar-char-variable* in the NAME= specifier becomes undefined, and the *scalar-char-variable* in the SEQUENTIAL=, DIRECT=, FORMATTED=, and UNFORMATTED= specifier are assigned the value UNKNOWN. Note that the *scalar-int-variable* in the NUMBER= specifier is always defined unless an error condition occurs during the execution of the INQUIRE statement. Note also that the specifier variables in the ACCESS=, FORM=, and BLANK= specifiers always become defined unless an error condition occurs during the execution of the INQUIRE statement.

The *inquire-spec-list* in an INQUIRE by unit statement must contain exactly one UNIT= specifier and must not contain a FILE= specifier. 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.

Execution of an INQUIRE by unit statement causes the specifier variables or array elements in the NUMBER= specifier, NAMED= specifier, FORM= specifier, ACCESS= specifier, SEQUENTIAL= specifier, DIRECT= specifier, NAME= specifier, FORMATTED= specifier, UNFORMATTED= specifier, RECL= specifier, NEXTREC= specifier, and BLANK= specifier to be assigned values only if the specified unit exists and if a file is connected to the unit; otherwise, the *scalar-int-variable* in the NUMBER= specifier is assigned the value −1, the *scalar-logical-variable* in the NAMED= specifier is assigned the value false, the *scalar-char-variable* in the FORM= specifier is undefined, the *scalar-char-variable* in the ACCESS= and FILE= specifiers are assigned the value UNDEFINED, and the *scalar-char-variable* in the SEQUENTIAL=, DIRECT=, FORMATTED=, and UNFORMATTED= specifiers are assigned the value UNKNOWN.

9.7 Restrictions on Function References and List Items.

A function reference must not appear in an expression anywhere in an input/output statement if such a reference causes another input/output statement to be executed. Note that restrictions in the evaluation of expressions (7.1.7) prohibit certain side effects.

9.8 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.

10 INPUT/OUTPUT EDITING

A format used in conjunction with an input/output statement provides information that directs the editing between the internal representation of data and the character strings of a record or a sequence of records in a file.

A format specifier (9.4.1.1) in an input/output statement may refer to a FORMAT statement or to a character expression that contains a format specification. A format specification provides explicit editing information. The format specifier also may be an asterisk (*) which indicates list-directed formatting (10.8), or a double asterisk (**) which indicates name-directed formatting (10.9).

10.1 Explicit Format Specification Methods. Explicit format specification may be given:

(1) In a FORMAT statement, or

(2) As the value of a character expression

10.1.1 FORMAT Statement.

R1001 *format-stmt*	**is**	FORMAT *format-specification*
R1002 *format-specification*	**is**	([*format-item-list*])

Constraint: The *format-stmt* must be labeled.

Constraint: The comma used to separate *format-items* in a *format-item-list* may be omitted as follows:

(1) Between a P edit descriptor and an immediately following F, E, EN, D, or G edit descriptor (10.6.5)

(2) Before or after a slash edit descriptor when the optional repeat specification is not present (10.6.2)

(3) Before or after a colon edit descriptor (10.6.3)

Note that, for source form purposes, the format specification is considered to be a from of character context (3.3.1).

10.1.2 Character Format Specification. A character expression used as a format specifier in a formatted input/output statement must contain a character string whose value constitutes a valid format specification. Note that the format specification begins with a left parenthesis and ends with a right parenthesis.

All character positions up to and including the final right parenthesis of the format specification must be defined at the time the input/output statement is executed, and must not become redefined or undefined during the execution of the statement. Character positions, if any, following the right parenthesis that ends the format specification need not be defined and may contain any character data with no effect on the format specification.

If the format specifier identifies a character array entity, 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 the subscript order value (6.2.4.2). However, if a format specifier refers to a character array element, the format specification must be contained entirely within that array element.

10.2 Form of a Format Item List.

R1003	*format-item*	**is** [*r*] *data-edit-desc*
		or *control-edit-desc*
		or *char-string-edit-desc*
		or [*r*] (*format-item-list*)
R1004	*r*	**is** *int-constant*

Constraint: *r* must be positive. It is called a **repeat specification.**

Constraint: The character underscore (_) is prohibited in an *int-constant* in a format specification.

Blank characters may precede the initial left parenthesis of the format specification. Additional blank characters may appear at any point within the format specification, with no effect on the format specification, except within a character string edit descriptor (10.7.1 and 10.7.2).

10.2.1 Edit Descriptors. An **edit descriptor** is used to specify the form of a record and to direct the editing between the characters in a record and internal representations of data. The internal representation of a datum corresponds to the internal representation of a constant of the corresponding type.

An edit descriptor is either a **data edit descriptor**, a **control edit descriptor**, or a **character string edit descriptor.**

R1005	*data-edit-desc*	**is** I *w* [. *m*]
		or F *w* . *d*
		or E *w* . *d* [E *e*]
		or EN *w* . *d* [E *e*]
		or G *w* . *d* [E *e*]
		or B *w*
		or L *w*
		or A [*w*]
		or D *w* . *d*
R1006	*w*	**is** *scalar-int-constant*
R1007	*m*	**is** *scalar-int-constant*
R1008	*d*	**is** *scalar-int-constant*
R1009	*e*	**is** *scalar-int-constant*

Constraint: *w* and *e* must be positive and *d* and *m* must be zero or positive.

Constraint: The value of *m*, *d*, and *e* may be restricted further by the value of *w*.

I, F, E, EN, D, G, B, L, and A indicate the manner of editing.

R1010	*control-edit-desc*	**is** *position-edit-desc*
		or [*r*] /
		or :
		or *sign-edit-desc*
		or *k* P
		or *blank-interp-edit-desc*
R1011	*k*	**is** *scalar-signed-int-constant*
R1012	*position-edit-desc*	**is** T *n*

	or TL *n*
	or TR *n*
	or *n* X
R1013 *n*	**is** *scalar-int-constant*
R1014 *sign-edit-desc*	**is** S
	or SP
	or SS
R1015 *blank-interp-edit-desc*	**is** BN
	or BZ

n must be positive.

In *k*P, *k* is called the **scale factor**.

T, TL, TR, X, slash, colon, S, SP, SS, P, BN, and BZ indicate the manner of editing.

R1016 *char-string-edit-desc*	**is** *char-constant*
	or *c* H *character* [*character*]...
R1017 *c*	**is** *int-constant*

Constraint: *c* must be positive.

Each character in a character string edit descriptor must be one of the characters capable of representation by the processor.

The character string edit descriptors provide constant data to be output, and are not valid for input.

Within a character constant, appearances of the delimiter character itself, apostrophe or quote, must be as consecutive pairs without intervening blanks. Each such pair represents a single occurrence of the delimiter character.

In the H edit descriptor, *c* specifies the number of characters following the H that comprise the descriptor.

10.2.2 Fields. A **field** is a part of a record that is read on input or written on output when format control encounters a data edit descriptor or a character string edit descriptor. The **field width** is the size in characters of the field.

10.3 Interaction Between Input/Output List and Format.

The beginning of formatted data transfer using a format specification 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 effective item in the input/output list, if one exists. Zero-sized arrays, zero-sized array sections, and implied-DO lists with iteration counts of zero are ignored in determining the next effective item (9.4.2).

If an input/output list specifies at least one list item, at least one data edit descriptor must exist in the format specification. Note that an empty format item list of the form ( ) may be used only if no input/output list items are specified; in this case, one input record is skipped or one output record containing no characters is written. Except for a format item preceded by a repeat specification *r*, a format specification is interpreted from left to right.

A format item preceded by a repeat specification is processed as a list of *r* items, each identical to the format item but without the repeat specification and separated by commas. Note that an omitted repeat specification is treated in the same way as a repeat specification whose value is one.

To each data edit descriptor interpreted in a format specification, there corresponds one effective item specified by the input/output list (9.4.2), except that an input/output list item of type complex requires the interpretation of two F, E, EN, D, or G edit descriptors. For each control edit descriptor or character 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 data edit descriptor in a format specification, it determines whether there is a corresponding effective 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 such item, format control terminates.

If format control encounters a colon edit descriptor in a format specification and another effective input/output list item is not specified, format control terminates.

If format control encounters the rightmost parenthesis of a complete format specification and another effective input/output list item is not specified, format control terminates. However, if another effective input/output 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 item list 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 data 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 (10.6.5.1), the sign control edit descriptors (10.6.4), or the blank interpretation edit descriptors (10.6.6).

10.4 Positioning by Format Control. After each data edit descriptor or character string edit descriptor is processed, the file is positioned after the last character read or written.

After each T, TL, TR, X, or slash edit descriptor is processed, the file is positioned as described in 10.6.1.

If format control reverts as described in 10.3, the file is positioned in a manner identical to the way it is positioned when a slash edit descriptor is processed (10.6.2).

During a read operation, any unprocessed characters of the record are skipped whenever the next record is read.

10.5 Data Edit Descriptors. Data edit descriptors cause the conversion of data to or from its internal representation. On input, the specified variable becomes defined. On output, the specified expression is evaluated.

10.5.1 Numeric Editing. The I, F, E, EN, D, and G edit descriptors are used to specify the input/output of integer, real, double 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 (9.3.4.6) and any BN or BZ blank control that is currently in effect for the unit (10.6.6). Plus signs may be omitted. A field containing only blanks is considered to be zero.

(2) On input, with F, E, EN, 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 input, single underscores may be inserted between otherwise adjacent digits of a constant to improve readability (4.3.1.1). Underscores do not affect the value of the constant.

(4) On output, the representation of a positive or zero internal value in the field may be prefixed with a plus, as controlled by the S, SP, and SS edit descriptors or the processor. 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.

(5) 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.

(6) 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*, EN*w*.*d*E*e*, or G*w*.*d*E*e* edit descriptor, the processor must 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.

(7) On output, the processor must not produce underscores in numeric fields.

10.5.1.1 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, 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.

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.

10.5.1.2 Real and Double Precision Editing. The F, E, EN, D, and G edit descriptors specify the editing of real, double precision, and complex data. An input/output list item corresponding to an F, E, EN, D, or G edit descriptor must be real, double precision, or complex.

10.5.1.2.1 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, including any blanks interpreted as zeros. 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, except for the interpretation of blanks

(3) D followed by zero or more blanks, followed by an optionally signed integer constant

An exponent containing a D 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.

10.5.1.2.2 E and D Editing. The E*w.d*, D*w.d*, and E*w.d*E*e* 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 and interpretation of the input field is the same as for F editing (10.5.1.2.1).

The form of the output field for a scale factor of zero is:

$[\pm]\,[0]\,.\,x_1x_2 \cdots x_d\,exp$

where:

$\pm$ signifies a plus or a minus.

$x_1x_2 \cdots x_d$ are the *d* most significant digits of the datum value after rounding.

exp is a decimal exponent having one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
E*w.d*	$\lvert exp \rvert \le 99$	$E\pm z_1z_2$ *or* $\pm 0z_1z_2$
	$99 < \lvert exp \rvert \le 999$	$\pm z_1z_2z_3$
E*w.d*E*e*	$\lvert exp \rvert \le 10^e - 1$	$E\pm z_1z_2 \cdots z_e$
D*w.d*	$\lvert exp \rvert \le 99$	$D\pm z_1z_2$ or $E\ \pm z_1z_2$ or $\pm 0z_1z_2$
	$99 < \lvert exp \rvert \le 999$	$\pm z_1z_2z_3$

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 E*w.d* and D*w.d* must not be used if $|exp| > 999$.

The scale factor *k* controls the decimal normalization (10.2.1). If $-d < k \le 0$, 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.

10.5.1.2.3 EN Editing. The EN edit descriptor will produce an output field of a real number in engineering notation such that the decimal exponent is divisible by three and the absolute value of the mantissa is greater than or equal to one and less than 1000, except when the output value is zero. The scale factor has no effect on output.

The forms of the edit descriptor are EN*w.d* and EN*w.d*E*e* indicating that the external field occupies *w* positions, the fractional part of which consists of *d* digits and the exponent part consists of *e* digits.

The form and interpretation of the input field is the same as for F editing (10.5.1.2.1), and the scale factor, if present, has an effect.

The form of the output field is:

$[\ \pm\ ]\ yyy.x_1x_2 \cdots x_d\, exp$

where:

$\pm$ signifies a plus or a minus.

yyy are the 1 to 3 decimal digits representative of the most significant digits of the value of the datum after rounding (*yyy* is an integer such that $1 \le yyy \le 999$ or $yyy = 0$).

$x_1x_2 \cdots x_d$ are the *d* next most significant digits of the value of the datum after rounding.

exp is a decimal exponent, divisible by three, of one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
EN*w.d*	$\lvert exp \rvert \le 99$	$E\pm z_1z_2$ *or* $\pm 0z_1z_2$
	$99 < \lvert exp \rvert \le 999$	$\pm z_1z_2z_3$
EN*w.d*E*e*	$\lvert exp \rvert \le 10^e - 1$	$E\pm z_1z_2 \cdots z_e$

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 form EN*w.d* must not be used if $\lvert exp \rvert > 999$.

Examples:

Internal Value	Output field Using EN12.3
6.421	6.421E+00
−.5	−500.000E−03
.00217	2.170E−03
4721.3	4.721E+03

10.5.1.2.4 G Editing. The G*w.d* and G*w.d*E*e* 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 form of the input field is the same as for F editing (10.5.1.2.1).

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 $0 < N < 0.1$ or $N \ge 10^d$, G*w.d* output editing is the same as *k*PE*w.d* output editing and G*w.d*E*e* output editing is the same as *k*PE*w*.*d*E*e* output editing, where *k* is the scale factor currently in effect. If $0.1 \le N < 10^d$ or *N* is identically 0, the scale factor has no effect, and the value of *N* determines the editing as follows:

Magnitude of Datum	Equivalent Conversion
$N = 0$	$F(w - n).(d - 1), n('b')$
$0.1 \le N < 1$	$F(w - n).d, n('b')$
$1 \le N < 10$	$F(w - n).(d - 1), n('b')$
.	.
.	.
.	.
$10^{d-2} \le N < 10^{d-1}$	$F(w - n).1, n('b')$
$10^{d-1} \le N < 10^d$	$F(w - n).0, n('b')$

where *b* is a blank. *n* is 4 for G*w*.*d* and *e* + 2 for G*w*.*d*E*e*.

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.

10.5.1.3 Complex Editing. A complex datum consists of a pair of separate real data; therefore, the editing is specified by two F, E, EN, 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. Control and character string edit descriptors may be processed between the two successive F, E, D, or G edit descriptors.

10.5.2 B Editing. The B*w* edit descriptor indicates that the field occupies *w* positions. The specified input/output list item must be of type bit.

The input field consists of *w* − 1 blanks and either a 0 or a 1, in any order. The output field consists of *w* − 1 blanks followed by either a 0 or a 1. The specifiers BZ and BN have no effect on bit editing.

10.5.3 L Editing. The L*w* edit descriptor indicates that the field occupies *w* positions. The specified input/output list item must be of type logical.

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, depending on whether the value of the internal datum is true or false, respectively.

10.5.4 A Editing. The A[*w*] edit descriptor is used with an input/output list item of type character.

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.

10.6 Control Edit Descriptors.

10.6 Control Edit Descriptors. A control edit descriptor does not cause the transfer of data nor the conversion of data to or from internal representation, but may affect the conversion performed by subsequent data edit descriptors.

10.6.1 Position 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. Note that an *n*X edit descriptor has the same effect as a TR*n* edit descriptor.

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 initially filled with blanks.

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 such that subsequent editing causes a replacement.

10.6.1.1 T, TL, and TR Editing. The T*n* edit descriptor indicates that the transmission of the next character to or from a record is to occur at the *n*th character position.

The TL*n* edit descriptor indicates that the transmission of the next character to or from the record is to occur at the character position *n* characters backward from the current position. However, if the current position is less than or equal to position *n*, the TL*n* 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*n* edit descriptor indicates that the transmission of the next character to or from the record is to occur at the character position *n* characters forward from the current position.

Note that *n* must be specified, and must be greater than zero.

10.6.1.2 X Editing. The *n*X edit descriptor indicates that the transmission of the next character to or from a record is to occur at the position *n* characters forward from the current position. Note that the *n* must be specified and must be greater than zero.

10.6.2 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. The repeat specification is optional on the slash edit descriptor. If it is not specified, a value of 1 is implied.

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.

10.6.3 Colon Editing. The colon edit descriptor terminates format control if there are no more effective items in the input/output list (9.4.2). The colon edit descriptor has no effect if there are more effective items in the input/output list.

10.6.4 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 has the option of producing 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 contains an optional 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 option of producing the plus is restored to the processor.

The S, SP, and SS edit descriptors affect only I, F, E, EN, 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.

10.6.5 P Editing. The *k*P edit descriptor sets the value of the scale factor to *k*. The scale factor may affect the editing of numeric quantities.

10.6.5.1 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, EN, D, and G edit descriptors until another P edit descriptor is encountered, and then a new scale factor is established. Note that reversion of format control (10.3) 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, EN, 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, EN, 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 significand (4.3.1.2) 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.

(5) On output, with EN editing, the scale factor has no effect.

10.6.6 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, nonleading blank characters are interpreted as zeros or are ignored, depending on the value of the BLANK= specifier (9.3.4.6) currently in effect for the unit. If a BN edit descriptor is encountered in a format specification, all nonleading 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 containing only blanks has the value zero. If a BZ edit descriptor is encountered in a format specification, all nonleading blank characters in succeeding numeric input fields are treated as zeros.

The BN and BZ edit descriptors affect only I, F, E, EN, D, and G editing during execution of an input statement. They have no effect during execution of an output statement.

10.7 Character String Edit Descriptors.

A character string edit descriptor must not be used on input.

10.7.1 Character Constant Edit Descriptor.

The character constant edit descriptor causes characters to be written from the enclosed characters of the edit descriptor itself, including blanks. Note that a delimiter is either an apostrophe or quote.

For a character constant edit descriptor, the width of the field is the number of characters contained in, but not including, the delimiting characters. Within the field, two consecutive delimiting characters with no intervening blanks are counted as a single character.

10.7.2 H Editing.

The *c*H edit descriptor causes character information to be written from the next *c* characters (including blanks) following the H of the *c*H edit descriptor in the *format-list* itself. The H edit descriptor must not be used on input. If a *c*H edit descriptor occurs within a character constant delimited by apostrophes and the H edit descriptor includes an apostrophe, the apostrophe must be represented by two consecutive apostrophes which are counted as one character in specifying *c*. If a *c*H edit descriptor occurs within a character constant delimited by quotes and the H edit descriptor includes a quote, the quote must be represented by two consecutive quotes which are counted as one character in specifying *c*.

10.8 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 appearances of the null value. 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 nonblank values or following the last nonblank value, where a nonblank value is a constant, an *r***c* form, or an *r** form.

10.8.1 List-Directed Input. Input forms acceptable to edit descriptors 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.

When the corresponding input list item is of type real or double precision, the input form is that of a numeric input field. A numeric input field is a field suitable for F editing (10.5.1.2.1) that 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 logical, the input form must not include slashes, blanks, or commas among the optional characters permitted for L editing.

When the corresponding list item is of type character, the input form consists of a character constant. Character constants may be continued from the end of one record to the beginning of the next record, but the end of record must not occur between a doubled apostrophe in an apostrophe-delimited constant, nor between a doubled quote in a quote-delimited constant.. 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.

If the corresponding input list item is of type character and:

(1) The character constant does not contain the characters blank, comma, or slash, and

(2) The datum does not cross a record boundary, that is, does not contain an end-of-record mark, and

(3) The first nonblank character is not a quotation mark or an apostrophe, and

(4) The leading characters are not numeric followed by an asterisk,

the delimiting apostrophes or quotation marks are not required. If the delimiters are omitted, the character constant is terminated by the first blank, comma, or slash character and apostrophes and quotation marks within the datum are not to be doubled.

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 (7.5.1.4).

10.8.1.1 Null Values. 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. Note that the end of a record following any other separator, with or without separating blanks, does not specify a null value. A null value has no effect on the definition status of the corresponding input list item.

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.

Any DO variable in the input list is defined as though enough null values had been supplied for any remaining input list items.

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

10.8.2 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 nondelimited character constants, the values are separated by (1) one or more blanks or (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.

Logical output constants are T for the value true and F for the value false.

Bit output constants are 1 for the value B'1' and 0 for the value B'0'.

Integer output constants are produced with the effect of an I*w* edit descriptor.

Real and double 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 $10^{d_1} \leq x < 10^{d_2}$. If the magnitude x is within this range, the constant is produced using 0PF*w.d*; otherwise, 1PE*w.d*E*e* is used.

For numeric outputs, reasonable processor-dependent integer values of *w*, *d*, and *e* are used for each of the cases involved. Note that underscores are not produced.

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 for a file opened without a DELIM= specifier (9.3.4.9) or with a DELIM= specifier (9.3.4.9) with a value of NONE:

(1) Are not delimited by apostrophes or quotation marks,

(2) Are not preceded or followed by a value separator,

(3) Have each internal apostrophe or quotation mark represented externally by one apostrophe or quotation mark, and

(4) 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.

Character constants produced for a file opened with a DELIM= specifier with a value of QUOTE are delimited by quotes, are preceded and followed by a value separator, and have each internal quote represented on the external medium by two quotes.

Character constants produced for a file opened with a DELIM= specifier with a value of APOSTROPHE are delimited by apostrophes, are preceded and followed by a value separator, and have each internal apostrophe represented on the external medium by two apostrophes.

If two or more successive values in an output record 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.

10.9 Name-Directed Formatting. The characters in one or more name-directed records constitute a sequence of name-value subsequences, each of which consists of a name followed by an equals and followed by one or more values and value separators. The equals may optionally be preceded or followed by zero, one, or more contiguous blanks. 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.

The name may be any name in the input/output list.

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

*r*c*
*r**

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

A value separator for name-directed formatting is the same as for list-directed (10.8) except that a value separator containing a slash must not immediately precede a value.

10.9.1 Name-Directed Input. Input for name-directed formatting consists of a sequence of zero or more name-value subsequences separated by value separators. In each name-value subsequence, the name must be the name of an input list item, optionally qualified as noted.

If a processor is capable of representing letters in both upper and lower case, the name may be in either case. Any subscripts or substring ranges appearing in the name must contain only integer constant expressions.

Within the input data, each name must correspond to a specific input list name. Subscripts within input list names must be integer constants. If an input list name is the name of an array, the name in the input record corresponding to it may be either the array name or the name of an element of that array, indicated by qualifying the array name with constant subscripts. If the input list name is the name of a variable of derived type, the name in the input record may be either the name of the variable or of one of its components, indicated by qualifying the variable name with the appropriate component name. Successive qualifications may be applied as appropriate to the shape and type of the variable represented.

The order of names in the input records need not match the order of the input list items. The input records need not contain all the names of the input list items. The definition status of any names from the input list that do not occur in the input record remains unchanged. The name in the input record may be preceded and followed by one or more optional blanks but must not contain embedded blanks.

The datum *c* is any input value acceptable to format specifications for a given type, except as noted. The form of the input value must be acceptable for the type of the input list item. The number and forms of the input values which may follow the equals in a name-value subsequence depend on the shape and type of the object represented by the name in the input record. When the name in the input record is the name of a scalar variable of an intrinsic type, the equals must not be followed by more than one value. This value must be of a form acceptable to format specifications for that type, except as noted. Blanks are never used as zeros, and embedded blanks are not permitted in constants except within character constants.

When the name in the input record is the name of an array of length *n*, at most *n* input values may follow the equals. The array element values must be specified in subscript order value.

The name-directed input statement is terminated by a slash encountered as a value separator during execution of a name-directed input statement after the assignment of the previous values.

When the corresponding input list item is of type real or double precision, the input form of the input value is that of a numeric input field. A numeric input field is a field suitable for F editing (10.5.1.2.1) that 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 of the input value 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 part 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 logical, the input form of the input value must not include either slashes, blanks, equals, or commas among the optional characters permitted for L editing (10.5.3).

When the corresponding list item is of type character, the input form of the input value consists of a nonempty string of characters enclosed in apostrophes or quotation marks. Each apostrophe within a character constant delimited by apostrophes must be represented by two consecutive apostrophes without an intervening blank or end of record. Each quotation mark within a character constant delimited by quotation marks must be represented by two consecutive quotation marks 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, equals, 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 (7.5.1.4).

If the corresponding list item is of type character and (1) the character constant does not contain the value separators blank, comma, slash, or equals, (2) the character constant does not cross a record boundary, (3) the first nonblank character is not a quotation mark or an apostrophe, and (4) the leading characters are not numeric followed by an asterisk, then the enclosing apostrophes or quotation marks are not required and apostrophes or quotation marks within the character constant are not to be doubled.

10.9.1.1 Null Values. A null value is specified by:

(1) *r** form

(2) Blanks between two consecutive value separators following an equals

(3) Zero or more blanks preceding the first value separator and following an equals, or

(4) Two consecutive nonblank value separators

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 must 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 a value separator, with or without intervening blanks, does not specify a null value.

10.9.1.2 Blanks. All blanks in a name-directed input record are considered to be part of some value separator except for:

(1) Blanks embedded in a character constant,

(2) Embedded blanks surrounding the real or imaginary part of a complex constant,

(3) Leading blanks following the equals unless followed immediately by a slash or comma, and

(4) Blanks between a name and the following equals.

10.9.2 Name-Directed Output. The form of the output produced is the same as that required for input, except as noted otherwise. If the processor is capable of representing letters in both upper and lower case, the name in the output is in upper case. With the exception of nondelimited character constants, the values are separated by (1) one or more blanks or (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. However, except for complex constants and character constants, the end of a record must not occur within a constant or a name, and blanks must not appear within a constant or a name.

Logical output constants are T for the value true and F for the value false.

Bit output constants are 1 for the value B'1' and 0 for the value B'0'.

Integer output constants are produced with the effect of an Iw edit descriptor.

Real and double 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 $10^{d_1} \le x < 10^{d_2}$. If the magnitude *x* is within this range, the constant is produced using 0PF*w.d*; otherwise, 1PE*w.d*E*e* is used.

For numeric output, reasonable processor-dependent integer 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 for a file opened without a DELIM= specifier (9.3.4.9) or with a DELIM= specifier with a value of NONE:

(1) Are not delimited by apostrophes or quotation marks

(2) Are not preceded or followed by a value separator

(3) Have each internal apostrophe or quotation mark and represented externally by one apostrophe or quotation mark

(4) 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.

Character constants produced for a file opened with a DELIM= specifier with a value of QUOTE are delimited by quotes, are preceded and followed by a value separator, and have each internal quote represented on the external medium by two quotes.

Character constants produced for a file opened with a DELIM= specifier with a value of APOSTROPHE are delimited by apostrophes, are preceded and followed by a value separator, and have each internal apostrophe represented on the external medium by two apostrophes.

If two or more successive values in an array 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.

The name of each output list item is placed in the output record followed by an equals and one or more values of the output list item.

A slash will be produced by name-directed formatting to indicate the end of the name-directed formatting.

A null value will not be produced by name-directed formatting.

Each output record begins with a blank character to provide carriage control when the record is printed.

11 PROGRAM UNITS

The terms and basic concepts of program unit were introduced in 2.2. An external program unit may be a main program, procedure subprogram, module subprogram, or block data subprogram. An internal program unit is a procedure subprogram.

This section describes all of these program units except procedure subprograms, which are described in Section 12.

11.1 Main Program.

R200	*main-program*	**is** [*program-stmt*] *program-unit-body* *end-program-stmt*
R1101	*program-stmt*	**is** PROGRAM *program-name*
R1102	*end-program-stmt*	**is** END [PROGRAM [*program-name*]]

Constraint: The *program-name* may be included in the *end-program-stmt* only if the optional *program-stmt* is used and, if included, must be identical to the *program-name* specified in the *program-stmt*.

The **program name** is global to the executable program, and must not be the same as the name of any other external program unit, external procedure, or common block in the executable program, nor the same as any local name in the main program.

11.1.1 Main Program Specifications. The specifications in the main program must not include an OPTIONAL statement, an INTENT statement, a PUBLIC statement, a PRIVATE statement, or the equivalent attributes (5.1.2). A SAVE statement has no effect in a main program.

11.1.2 Main Program Executable Part. The sequence of *execution-part* statements specifies the actions of the main program during program execution. Execution of an executable program (R201) begins with the first executable construct of the main program. A main program *execution-part* statement may be any of those listed in syntax rules R214, R217, and R218 of Section 2.1, except a RETURN statement or an ENTRY statement.

A main program must not be recursive; that is, a reference to it must not appear in any program unit in the executable program, including itself.

Execution of an executable program ends with execution of the END PROGRAM statement of the main program, with the signalling of a condition in the main program for which there is no handler, or with execution of a STOP statement in any program unit of the executable program.

11.1.3 Main Program Internal Procedures. Any definitions of procedures internal to the main program follow the CONTAINS statement. Internal procedures are described in Section 12. The main program is called the **host** of its internal procedures, but not of procedures internal to them.

11.2 Procedure Subprograms.

Procedure subprograms are described in Section 12.

11.3 Module Subprograms. A module contains a set of specifications and definitions that are to be used by other program units.

R200	*module-subprogram*	**is** *module-stmt* *program-unit-body* *end-module-stmt*
R1103	*module-stmt*	**is** MODULE *module-name*
R1104	*end-module-stmt*	**is** END [MODULE [*module-name*]]

Constraint: If the *module-name* is specified in the *end-module-stmt*, it must be identical to the *module-name* specified in the *module-stmt*.

The module name is global to the executable program, and must not be the same as the name of any other external program unit, external procedure, or common block in the executable program, nor the same as any local name in the module subprogram.

A USE statement specifying a module name is a **module reference**. At the time a module reference is processed, the public portions of the specified module subprogram must be available. A module subprogram must not reference itself, either directly or indirectly.

The accessibility, public or private, of specifications and definitions in a module to a program unit making reference to the module may be controlled in both the module and the program unit making the reference. In the module, the PRIVATE statement, the PUBLIC statement (5.2.3), and the equivalent attributes (5.1.2.2) are used to control the accessibility of module entities outside the module.

In a program unit making reference to a module, options on the USE statement may be used to further limit the availability, to that referencing program unit, of the public entities in the module.

A module specification may be any of those listed in syntax rule R216 of Section 2.1, except an INTENT statement, an OPTIONAL statement, and the equivalent INTENT and OPTIONAL attributes.

11.3.1 The USE Statement. The **USE statement** provides the means by which a program unit accesses entities in a module subprogram or, in the case of an internal procedure, in its host.

R1105	*use-stmt*	**is** USE [*module-name*] [[,] *all-clause*] **or** USE [*module-name*] [,] ONLY ([*only-list*])
R1106	*all-clause*	**is** ALL ([*rename-list*]) **or** ALL [([*rename-list*])] EXCEPT (*except-list*)
R1107	*rename*	**is** *use-name* => *local-name*
R1108	*except*	**is** *use-name*
R1109	*only*	**is** *use-name* [=> *local-name*]
R1110	*use-name*	**is** *variable-name* **or** *procedure-name* **or** *type-name* **or** *condition-name* **or** *constant-name*

The USE statement with neither the ALL option nor the ONLY option implies the ALL option with an empty *rename-list*.

In a USE statement, the module name must not be omitted except in an internal procedure and, in this case, access is provided to entities in its host. If an internal procedure has no such USE statement, it is as if it contained a USE ALL ( ) statement. A program unit may contain more than one USE statement with the module name omitted or more than one USE statement for a single module name.

Each *use-name* must be the name of a public entity in the module or an entity in the host. If a *local-name* appears in a *rename-list* or an *only-list*, it is the local name for the entity *original-name*; otherwise, the local name is the *use-name*.

A USE statement with the ONLY option provides access to those entities whose names appear as use names in its *only-list*. Such entities are **explicitly accessible**.

A USE statement with an explicit ALL option provides access to those entities whose names appear as use names in its *rename-list*. Such names are explicitly accessible. In addition, a USE statement with an implicit or explicit ALL option provides access to certain entities that are not explicitly listed. Entities accessible by this means are **implicitly accessible**. All public eligible entities in the accessed program unit that are simple variables, symbolic constants, types, procedures, or conditions are implicitly accessible except:

(1) Any entity named in an *except-list*,

(2) Any entity named in the *rename-list* (these are explicitly accessible),

(3) In the case of a USE statement in a procedure, any entity whose name is the same as the name of a dummy argument of the procedure, and

(4) In the case of a USE statement in a function subprogram, any entity whose name is the same as the name of the function or its result variable.

An entity may be made implicitly accessible with the same name as an entity made implicitly or explicitly accessible from a different host or module if the entity is the same one (ultimately acquired from the same source). In a program unit, two or more entities given implicit accessibility by USE statements may have the same name provided no entity is accessed by this name in the program unit. Except for this, the local name of any entity given accessibility by a USE statement must differ from the local names of all other entities accessible from the program unit through USE statements and otherwise. Note that an entity may be accessed by more than one local name.

A local name of an entity given accessibility by a USE statement in a program unit may appear in a PRIVATE or PUBLIC statement, but in no other specification statement in the program unit.

If a program unit has implicit accessibility to entities of a second program unit, this includes the entities to which the second program unit has implicit accessibility.

Examples:

```
USE STATS_LIB
```

provides access to all public entities in the module STATS_LIB.

```
USE ONLY ()
```

ensures that an internal procedure has no access to entities in its host except through argument association and COMMON association.

```
USE MATH_LIB, ALL EXCEPT (ML_X15)
```

makes accessible all public entities in the module MATH_LIB except ML_X15.

```
USE MATH_LIB; USE STATS_LIB, ALL (PROD => SPROD)
```

makes all public entities in both MATH__LIB and STATS__LIB accessible. If MATH__LIB contains an entity called PROD, it is accessible by its own name while the entity PROD of STATS__LIB is accessible by the name SPROD. Both modules may contain an entity called SUMM, for example, if SUMM does not appear in the program unit containing the USE statements and SUMM is not declared in a type statement in the program unit.

11.3.2 Examples of Modules.

11.3.2.1 Identical Common Blocks. A common block and all its associated specification statements may be placed in a module named, for example, COMMON and accessed by a USE statement of the form

```
USE COMMON
```

that accesses the whole module without any renaming. This ensures that all instances of the common block are identical. Module COMMON could contain more than one common block.

11.3.2.2 Global Data. A module may contain just data objects, for example

```
MODULE DATA_MODULE
REAL A(10), B, C(20,20)
INTEGER, INITIAL :: I=0
INTEGER, PARAMETER :: J=10
COMPLEX D(J,J)
END MODULE
```

Note that data objects made global in this manner may have any combination of data types.

Access to some of these may be made by a USE statement with the ONLY option, such as:

```
USE DATA_MODULE, ONLY (A, B, D)
```

and access to all of them may be made by the following USE statement

```
USE DATA_MODULE
```

Access to all of them with some renaming to avoid name conflicts may be made by:

```
USE DATA_MODULE, ALL (A => AMODULE, D => DMODULE)
```

11.3.2.3 Data Structures. A derived type may be defined in a module for use in a number of external program units. This is the only way to access the same type definition in more than one program unit. For example:

```
MODULE SPARSE
TYPE NONZERO
   REAL A
   INTEGER  I, J
END TYPE
END MODULE
```

defines a type consisting of a real component and two integer components for holding the numerical value of a nonzero matrix element and its row and column indices.

11.3.2.4 Global Allocatable Arrays. Many programs need large global allocatable arrays whose sizes are not known before program execution. A simple form for such a program is:

```
PROGRAM GLOBAL_WORK
CALL CONFIGURE_ARRAYS        ! PERFORM THE APPROPRIATE ALLOCATIONS
CALL COMPUTE                 ! USE THE ARRAYS IN COMPUTATIONS
```

```
END PROGRAM GLOBAL_WORK

MODULE WORK_ARRAYS         ! AN EXAMPLE SET OF WORK ARRAYS
INTEGER  N
REAL, ALLOCATABLE, SAVE :: A(:), B(:,:), C(:,:,:)
END MODULE WORK_ARRAYS

SUBROUTINE CONFIGURE_ARRAYS  ! PROCESS TO SET UP WORK ARRAYS
USE WORK_ARRAYS
READ (INPUT,*)  N
ALLOCATE ( A(N), B(N,N), C(N,N,2*N) )
END SUBROUTINE CONFIGURE_ARRAYS

SUBROUTINE COMPUTE
USE WORK_ARRAYS
!  COMPUTATIONS INVOLVING ARRAYS A, B, AND C
END SUBROUTINE COMPUTE
```

Typically, many procedures need access to the work arrays, and all such procedures would contain the statement

```
USE WORK_ARRAYS
```

11.3.2.5 Procedure Libraries. Interfaces to external procedures in a library may be gathered into a module. This permits the use of keyword and optional arguments, and allows static checking of the references. Different versions may be constructed for different applications, using keywords in common use in each application. An example is the following library module:

```
MODULE LIBRARY_LLS
   INTERFACE
      SUBROUTINE LLS (X, A, F, FLAG)
      REAL (*, *) X (:, :)
      REAL (*, *), ARRAY (SIZE (X, 2)) :: A, F
      INTEGER FLAG
   END INTERFACE
END MODULE
```

This module allows the subroutine LLS to be invoked:

```
USE LIBRARY_LLS
   ...
CALL LLS (X = ABC, A = D, F = XX, FLAG = IFLAG)
   ...
```

11.3.2.6 Operator Extensions. To extend the operator + to have the meaning .BOR. for bit operands, the following may be written:

```
MODULE BIT_PLUS
   FUNCTION OR(B1,B2) OPERATOR(+)
   BIT :: OR, B1, B2
   OR = B1 .BOR. B2
   END FUNCTION
END MODULE
```

A module might contain several such functions. If the operation is written in a language other than Fortran, it may be written as an external function and its procedure interface placed in the module.

11.4 Data Abstraction.

A module may encapsulate a derived-type definition and all the operations on values of this type. An example is given in Appendix C for set operations.

11.5 Block Data Subprograms.

A block data subprogram is used to provide initial values for data entities in named common blocks.

R1111	*block-data-subprogram*	is	*block-data-stmt* *program-unit-body* *end-block-data-stmt*
R1112	*block-data-stmt*	is	BLOCK DATA [*block-data-name*]
R1113	*end-block-data-stmt*	is	END [BLOCK DATA [*block-data-name*]]

Constraint: The *block-data-name* may be included in the *end-block-data-stmt* only if it was provided in the *block-data-stmt* and, if included, must be identical to the *block-data-name* in the *block-data-stmt*.

The specifications of a block data subprogram may contain only the following statements: type declaration, IMPLICIT, PARAMETER, SAVE, COMMON, DATA, DIMENSION, and EQUIVALENCE.

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 objects initially defined in a single block data subprogram. Note, therefore, that the primary constituents of a block data subprogram are type declarations of common block entities, COMMON statements, and DATA statements.

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 an executable program.

There must not be more than one unnamed block data subprogram in an executable program.

12 PROCEDURES

The concept of a procedure was introduced in 2.2.3. This section contains a complete description of procedures. The action specified by a procedure is performed when the procedure is invoked by execution of a reference to it. The reference may identify, as actual arguments, entities that are associated during execution of the procedure reference with corresponding dummy arguments in the procedure definition.

12.1 Procedure Classifications. A procedure is classified according to the form of its reference and the way it is defined.

12.1.1 Procedure Classification by Reference. The definition of a procedure specifies it to be a function or a subroutine. A reference to a function appears as a primary within an expression. A reference to a subroutine is a CALL statement or a defined assignment statement.

A procedure is classified as **elemental** if its reference to a subroutine is an elemental reference (12.4.3, 12.4.5).

12.1.2 Procedure Classification by Means of Definition. A procedure is either an intrinsic procedure, an external or internal procedure, a dummy procedure, or a statement function.

12.1.2.1 Intrinsic Procedures. A procedure that is provided as an inherent part of the processor is an **intrinsic procedure**.

12.1.2.2 External and Internal Procedures. A procedure that is defined by a procedure subprogram is either an external procedure or an internal procedure. If the procedure subprogram is not contained in another program unit, the procedure is an **external procedure**. If the procedure subprogram is contained in another program unit, the procedure is an **internal procedure**. Means other than Fortran may also be used to define an external or internal procedure.

If a procedure subprogram contains one or more ENTRY statements, it defines a procedure for each ENTRY statement and a procedure for the SUBROUTINE or FUNCTION statement.

12.1.2.3 Dummy Procedures. A dummy argument that is specified as a procedure or appears in a procedure reference is a **dummy procedure**.

12.1.2.4 Statement Functions. A function that is defined by a single statement is a **statement function**.

12.2 Characteristics of Procedures. The **characteristics** of a procedure consist of the classification of the procedure as a function or subroutine, the characteristics of its arguments, and the characteristics of its result value if it is a function.

12.2.1 Characteristics of Dummy Arguments. Each dummy argument is either a dummy data object, a dummy procedure, a dummy condition, or an asterisk. A dummy argument may be specified to have the OPTIONAL attribute. This attribute means that the dummy argument need not be associated with an actual argument for any particular reference to the procedure.

12.2.1.1 Characteristics of Dummy Data Objects. The characteristics of a dummy data object consists of its type, type parameters (if any), shape, intent (5.1.2.3, 5.2.1), optionality (5.1.2.7, 5.2.2), and whether it is allocatable (5.1.2.4.3). If a type parameter or a bound of an array is an expression, the exact dependence on other entities is a characteristic. If shape, size, or type parameters are assumed, these are characteristics.

12.2.1.2 Characteristics of Dummy Procedures. The characteristics of a dummy procedure consist of the explicitness of its interface (12.3.1), the characteristics of the procedure if the interface is explicit, and its optionality (5.1.2.7, 5.2.2).

12.2.1.3 Characteristics of Dummy Conditions. The only characteristic of a dummy condition is its optionality (5.1.2.7, 5.2.2).

12.2.1.4 Characteristics of Asterisk Dummy Arguments. An asterisk as a dummy argument has no characteristics.

12.2.2 Characteristics of Function Results. The characteristics of a function result consist of its type, type parameters (if any), shape, and whether it is allocatable. Where a type parameter or bound of an array is an expression, the exact dependence on other entities is a characteristic. If the length of a character data object is assumed, this is a characteristic.

12.3 Procedure Interface.

The **interface** of a procedure determines the forms of reference through which it may be invoked. The interface consists of the characteristics of the procedure, the name of the procedure, the name of each dummy argument, the operator (if any) by which a reference to a function may appear, and whether or not a reference to a subroutine may be implied in a defined assignment statement. The characteristics of a procedure are fixed, but the remainder of the interface may differ in different program units.

12.3.1 Implicit and Explicit Interfaces. If a procedure is accessible in a program unit, its interface is either **explicit** or **implicit** in that program unit. The interface of an internal procedure or intrinsic procedure is always explicit. For example, the subroutine LLS of 11.3.2.5 has an explicit interface. The interface of an external procedure or dummy procedure is explicit if an interface block (12.3.2.1) for the procedure is supplied and implicit otherwise. For example, when the function OR of 11.3.2.6 is written as an external function in another language, it has an explicit interface when the module BIT_PLUS is used to supply an interface block for it, rather than the internal function definition. The interface of a statement function is always implicit.

12.3.1.1 Explicit Interface. A procedure must have an explicit interface in a program unit if any of the following is true:

(1) A reference to the procedure appears:

 (a) With a keyword argument (12.4.1)

 (b) As a defined assignment (subroutines only)

 (c) In an expression as a defined operator (functions only)

 (d) As an elemental reference

(2) The procedure has:

 (a) An optional dummy argument

 (b) An array-valued result (functions only)

(c) An allocatable result (functions only)

(d) A dummy argument that is an assumed-shape or allocatable array

(e) A dummy argument with assumed type parameters other than character length

(f) A result whose type parameter values are neither assumed length (character type only) nor constant.

(3) Another procedure having the same name is accessible

12.3.1.2 Implicit Interface. An actual argument may be sequence associated (12.4.1.5) with its dummy argument if its interface is implicit.

12.3.2 Specification of the Procedure Interface. The interface for an internal, external, or dummy procedure is specified by a FUNCTION, SUBROUTINE, or ENTRY statement and by specification statements for the dummy arguments and the result of a function. These statements may appear in the procedure definition, in an interface block, or both.

12.3.2.1 Procedure Interface Block.

R1201	*interface-block*	**is**	*interface-stmt* *interface-specification* *end-interface-stmt*
R1202	*interface-stmt*	**is**	INTERFACE
R1203	*end-interface-stmt*	**is**	END INTERFACE
R1204	*interface-specification*	**is**	*interface-header* [*use-stmt*]... [*implicit-part*]... [*declaration-part*]...
R1205	*interface-header*	**is**	*function-stmt*
		or	*subroutine-stmt*.

An interface block in a program unit specifies the interface of the procedure named in the FUNCTION or SUBROUTINE statement in the interface block. The statements are interpreted as if they were the leading statements of an external procedure, except the interface block may contain a USE statement accessing the host program unit. For example,

```
IMPLICIT NONE
   INTERFACE
      FUNCTION INVERSE (A)
   END INTERFACE
   ...
```

is a valid fragment of code because the FUNCTION statement is interpreted as if it were the leading statement of an external function, so the default implicit typing rules are assumed.

An interface block that names as a procedure a dummy argument of the host program unit specifies that dummy argument to be a procedure with the specified interface. A dummy argument must not be so named more than once. Such a dummy argument may be specified in an OPTIONAL statement or with an OPTIONAL attribute in the host but must not appear in any other specification statement in the host. For example,

```
SUBROUTINE INVERSE (A, FN)
   REAL A
   INTERFACE
```

```
      FUNCTION FN (B)
      REAL FN, B
      ...
   END INTERFACE
   ...
```

specifies a subroutine whose second argument is a real function with a single real argument.

An interface block that does not name as a procedure a dummy argument of the host program unit specifies an interface to an external procedure. In a module, the name of the external procedure may appear in a PUBLIC or PRIVATE statement or be given the equivalent attribute, but must not appear in any other specification statement in the host.

The characteristics (12.2) of the procedure itself must be identical with those specified by the interface block. The presence of the block does not require the availability of the procedure until it is invoked.

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

R1206	*external-stmt*	**is** EXTERNAL *external-name-list*
R1207	*external-name*	**is** *external-procedure-name*
		or *dummy-arg-name*
		or *block-data-name*

The appearance of the name of a dummy argument in an EXTERNAL statement specifies that the dummy argument is a dummy procedure.

The appearance in an EXTERNAL statement of a name that is not the name of a dummy argument specifies that the name is the name of an external procedure or block data subprogram.

Only one appearance of a symbolic name in all of the EXTERNAL statements in any one sequence of declaration part statements is permitted.

12.3.2.3 INTRINSIC Statement. An INTRINSIC statement is used to specify a symbolic name as representing an intrinsic procedure (Section 13) or an intrinsic condition. It also permits a name that represents a specific intrinsic function to be used as an actual argument.

R1208	*intrinsic-stmt*	**is** INTRINSIC *intrinsic-name-list*
R1209	*intrinsic-name*	**is** *intrinsic-procedure-name*
		or *intrinsic-condition-name*

The appearance of a name in an INTRINSIC statement confirms that the name is the name of an intrinsic procedure or an intrinsic condition. The appearance of a generic function name (13.1) 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 in any one sequence declaration part statements is permitted. Note that a symbolic name must not appear in both an EXTERNAL and an INTRINSIC statement in the same sequence of declaration-part statements.

12.3.2.4 Implicit Interface Specification. In a program unit where the interface of a function is implicit, the type and type parameters of the function result are specified by implicit or explicit type specification of the function name. The type, type parameters, and shape of dummy arguments of a procedure referenced from a program unit where the interface of a procedure is implicit are assumed to be such that the actual arguments are consistent with the characteristics of the dummy arguments.

12.4 Procedure Reference.

The form of a procedure reference is dependent on the interface of the procedure, but is independent of the means by which the procedure is defined. The form of procedure references are:

R1210 *function-reference* **is** *function-name* ([*actual-arg-spec-list*])

Constraint: The *actual-arg-spec-list* for a function reference must not contain an *alt-return-spec.*

R1211 *call-stmt* **is** CALL *subroutine-name* [([*actual-arg-spec-list*])]

12.4.1 Actual Argument List.

R1212 *actual-arg-spec* **is** [*keyword* =] *actual-arg*

R1213 *keyword* **is** *dummy-arg-name*

R1214 *actual-arg* **is** *expr*
or *variable*
or *procedure-name*
or *condition-name*
or *alt-return-spec*

R1215 *alt-return-spec* is * *label*

Constraint: The *keyword* may be omitted from an *actual-arg-spec* only if the *keyword* has been omitted from each preceding *actual-arg-spec* in the argument list.

Constraint: Each *keyword* must be the name of a dummy argument in the interface of the procedure.

In either a subroutine reference or a function reference, the actual argument list identifies the correspondence between the actual arguments supplied and the dummy arguments of the procedure. In the absence of a keyword, an actual argument is associated with the dummy argument occupying the corresponding position in the dummy argument list; i.e., the first actual argument is associated with the first dummy argument, the second actual argument is associated with the second dummy argument, etc. If a keyword is present, the actual argument is associated with the dummy argument whose name is the same as the keyword. Exactly one actual argument must be associated with each nonoptional dummy argument. At most one actual argument may be associated with each optional argument. Each actual argument must be associated with a dummy argument. For example, the procedure

```
SUBROUTINE SOLVE (FUNCT, SOLUTION, METHOD, STRATEGY, PRINT)
   INTERFACE
      FUNCTION FUNCT (X)
      REAL FUNCT, X
   END INTERFACE
   REAL SOLUTION
   INTEGER, OPTIONAL :: METHOD, STRATEGY, PRINT
   ...
```

may be invoked with

```
CALL SOLVE (FUN, SOL, PRINT = 6)
```

12.4.1.1 Arguments Associated with Dummy Data Objects. If a dummy argument is a dummy data object, the associated actual argument must be an expression of the same type or a data object of the same type. The type parameter values of the actual argument, if any, must agree with or be assumed by the dummy argument. The shape of the actual argument must agree with or be assumed by the dummy argument except when a procedure reference is elemental (12.4.3, 12.4.5) or when the actual argument is sequence associated with the dummy argument (12.4.1.5). Each element of an array-valued actual argument or of a sequence in a sequence association (12.4.1.5) is associated with the element of the dummy array that has the same position in subscript order value (6.2.4.2).

If the intent of a dummy argument is OUT or INOUT, the actual argument must be definable. If the intent of a dummy argument is OUT and it or any part of it is not defined during the execution of the procedure, the corresponding actual argument or part of it becomes undefined.

12.4.1.2 Arguments Associated with Dummy Procedures. If a dummy argument is a dummy procedure, the associated actual argument must be the name of an external, internal, dummy, or intrinsic procedure. The only intrinsic procedures permitted are those listed in 13.8.17 and not marked with a bullet (•). The actual argument name must be one for which exactly one procedure is accessible in the invoking program unit. (A specific intrinsic function and a generic intrinsic function of the same name are considered to be one procedure.) The actual argument procedure must not have dummy arguments with assumed type parameters other than character assumed lengths.

If the interface of the dummy procedure is explicit, the characteristics of the associated procedure must be the same as the characteristics of the dummy procedure (12.2).

If the interface of the dummy procedure is implicit and either the name of the dummy procedure is explicitly typed or the procedure is referenced as a function, the dummy procedure must not be referenced as a subroutine and the actual argument must be a function or dummy procedure.

If the interface of the dummy procedure is implicit and a reference to the procedure appears as a subroutine reference, the actual argument must be a subroutine or dummy procedure.

12.4.1.3 Arguments Associated with Dummy Conditions. If a dummy argument is a dummy condition, the associated actual argument must be a condition name.

12.4.1.4 Arguments Associated with Alternate Return Indicators. If a dummy argument is an asterisk (12.5.2.3), the associated actual argument must be an alternate return specifier. The label in the alternate return specifier must identify an executable construct in the program unit containing the procedure reference.

12.4.1.5 Sequence Association. An actual argument represents an **element sequence** if it is a whole array name, array element name, or array element substring name and the array is neither a dummy argument that is not sequence associated, a ranged array, nor an alias array. If the actual argument is a whole array name, the element sequence consists of the elements in subscript order value. If the actual argument is an array element name, the element sequence consists of that array element and each element that follows it in subscript order value. If the actual argument is an array element substring name, the element sequence consists of the character storage units beginning with the first storage unit in that array element substring and continuing to the end of the array. The character storage units are viewed as elements consisting of consecutive groups of character storage units the length of the array element substring. Thus, the first such element is the array element substring itself. Note that some of the elements in the element sequence may consist of storage units from different elements of the original array.

If the interface for a procedure reference is implicit, the actual argument represents an element sequence, and the corresponding dummy argument is an array-valued data object that is neither allocatable nor assumed shape, the actual argument is **sequence associated** with the dummy argument. The rank and shape of the actual argument need not agree with the rank and shape of the dummy argument, but the number of elements in the dummy argument must not exceed the number of elements in the element sequence of the actual argument. If the dummy argument is assumed size, the number of elements in the dummy argument is exactly the number of elements in the element sequence.

12.4.2 Function Reference. A function is invoked during expression evaluation by a function reference or by defined operations (7.1.3). When it is invoked, all actual argument expressions are evaluated, then the arguments are associated, and then the function is executed. When execution of the function is complete, the value of the function result is available for use in the expression that caused the function to be invoked.

12.4.3 Elemental Function Reference. A reference to a function is an **elemental reference** if the interface for the function is explicit, if its dummy arguments and result are all scalar data objects, and if the type parameters of the result are independent of the values of the actual arguments. Arguments to such a reference may be arrays, provided all array arguments have the same shape. The result has the same shape as the array arguments and the value of each element in the result is obtained by evaluating the function using the scalar arguments and the corresponding elements of the array arguments. For example, if X and Y are arrays of shape [*m*, *n*],

```
MAX (X, 0.0, Y)
```

is an array expression of shape [*m*, *n*] whose elements have values

$$\text{MAX}\,(\text{X}\,(i, j), 0.0, \text{Y}\,(i, j)),\ i = 1, 2, ..., m,\ j = 1, 2, ..., n$$

The result must not depend on the order in which these references are made.

For example, the reference to the procedure

```
FUNCTION SCALE (A)
   READ (*, *) FACTOR
   SCALE = A * FACTOR
END
```

must not be an elemental reference.

A function reference is not interpreted as being such an elemental reference if it may be interpreted as a nonelemental reference to a function with the same name whose interface is explicit in the program unit containing the reference. For example, the expression SCALE ([1 : 10]) would not be interpreted as an elemental reference if a function SCALE with one integer argument of rank one is accessible.

12.4.4 Subroutine Reference. A subroutine is invoked by execution of a CALL statement or defined assignment statement (7.5.1.3). When a subroutine is invoked, all actual argument expressions are evaluated, then the arguments are associated, and then the subroutine is executed. When the action specified by the subroutine is completed, execution of the CALL statement or defined assignment statement is also completed. If a CALL statement includes one or more alternate return specifiers among its arguments, control may be transferred to one of the statements indicated, depending on the action specified by the subroutine.

12.4.5 Elemental Assignment. A reference to an assignment subroutine may be an elemental reference in a defined assignment statement if its dummy arguments are scalar and the type parameters of the first dummy argument are independent of the value of the second dummy argument. In such a reference, the first actual argument is array valued and the second is of the same shape or is scalar. The subroutine is invoked once for each element of the first actual argument, using the corresponding element of the second actual argument or its scalar value. The result must not depend on the order in which these invocations are made. An assignment is not interpreted as an elemental assignment if it may be interpreted as a nonelemental assignment. An example of a subroutine that may be invoked as an elemental assignment is:

```
SUBROUTINE BITLOG (B, L) ASSIGNMENT
   BIT B
   LOGICAL L
   B = BITL (L)
END
```

12.5 Procedure Definition.

12.5.1 Intrinsic Procedure Definition. Intrinsic procedures are defined as an inherent part of the processor. A standard-conforming processor must include the intrinsic procedures described in Section 13, but may include others. However, a standard-conforming program must not make use of intrinsic procedures other than those described in Section 13.

12.5.2 Procedures Defined by Procedure Subprograms. When a procedure defined by a procedure subprogram is invoked, an instance (12.5.2.4) of the procedure subprogram is created and executed. Execution begins with the first executable construct following the FUNCTION, SUBROUTINE, or ENTRY statement specifying the name of the procedure invoked.

12.5.2.1 Effects of Intent on Procedure Subprograms. The intent of dummy data objects limits the way in which they may be used in a procedure subprogram. A dummy data object having intent IN may not be defined or redefined by the procedure. A dummy data object having intent OUT is initially undefined in the procedure. A dummy data object with intent INOUT may be referenced or be defined. A dummy data object whose intent is neither specified nor implied by the presence of the OPERATOR or ASSIGNMENT option is subject to the limitations of the data entity that is the associated actual argument. That is, a reference to the dummy data object may appear if the actual argument is defined and may be defined if the actual argument is definable.

12.5.2.2 Function Subprogram.

R207	*function-subprogram*	**is** *function-stmt* *program-unit-body* *end-function-stmt*
R1216	*function-stmt*	**is** [*prefix*] FUNCTION *function-name* ([*dummy-arg-name-list*]) [*suffix*]
R1217	*prefix*	**is** *type-spec* [RECURSIVE] **or** RECURSIVE [*type-spec*]
R1218	*suffix*	**is** RESULT (*result-name*) [OPERATOR (*defined-operator*)] **or** OPERATOR (*defined-operator*) [RESULT (*result-name*)]
R1219	*end-function-stmt*	**is** END [FUNCTION [*function-name*]]

Constraint: FUNCTION must be present on the *end-function-stmt* of an internal function.

Constraint: If *function-name* is supplied on the *end-function-stmt*, it must agree with the *function-name* on the *function-stmt*.

The type of a function subprogram may be specified by a type specification in the FUNCTION statement or by the function name appearing in a type statement in the declaration part of the function subprogram. It may not be specified both ways. If it is not specified either way, it is determined by the implicit typing rules in force within the function subprogram. If the function result is array valued or allocatable, this must be specified by specifications of the function name within the function body.

The keyword RECURSIVE must be present if the function invokes itself, either directly or indirectly.

The name of the function is *function-name.*

If RESULT is specified, the name of the result variable of the function is *result-name.* Otherwise, it is *function-name*. The *result-name* must not appear in any specification statement.

If OPERATOR is specified, the interface for the procedure includes the ability to invoke it using a defined operator. This operator must be unary if the function has one dummy argument and binary if it has two dummy arguments; no other number of dummy arguments is permitted. The dummy arguments must be nonoptional dummy data objects with intent IN. If the intent of a dummy argument is not specified, the specification of OPERATOR causes it to have intent IN.

12.5.2.3 Subroutine Subprogram.

R1220 *subroutine-subprogram*	**is**	*subroutine-stmt*
		program-unit-body
		end-subroutine-stmt
R1221 *subroutine-stmt*	**is**	[RECURSIVE] SUBROUTINE *subroutine-name* □
		□ [(*dummy-arg-list*)] [ASSIGNMENT]
R1222 *dummy-arg*	**is**	*dummy-arg-name*
	or	*
R1223 *end-subroutine-stmt*	**is**	END [SUBROUTINE [*subroutine-name*]]

Constraint: SUBROUTINE must be present on the end of an internal subroutine.

Constraint: If *subroutine-name* is present on the *end-subroutine-stmt*, it must agree with the *subroutine-name* on the *subroutine-stmt*.

The keyword RECURSIVE must be present if the subroutine subprogram invokes itself, either directly or indirectly.

If ASSIGNMENT is specified, the subroutine may be referenced as an assignment statement and is called an **assignment subroutine**. The subroutine must have exactly two arguments which must be nonoptional dummy data objects. The first dummy argument must have intent OUT or INOUT. If its intent is not specified, it has intent OUT. The second dummy argument must have intent IN. If its intent is not specified, it has intent IN.

12.5.2.4 Instances of a Procedure Subprogram. When a function or subroutine defined by a procedure subprogram is invoked, an **instance** of that subprogram is created.

Each instance has an independent sequence of execution and an independent set of dummy arguments and nonsaved data objects. If an internal procedure or statement function contained in the subprogram is invoked directly from an instance of the subprogram or a procedure having access by explicit or implicit USE statements to the entities of that instance, then the created instance of that internal procedure or statement function also has access by explicit or implicit USE statements to the entities of that instance of the host subprogram. Similarly, if the internal procedure is supplied as an actual argument from an instance of the subprogram or a procedure having access by explicit or implicit USE statements to the entities of that instance, then the instance of that internal procedure created by invoking the associated dummy procedure also has access by explicit or implicit USE statements to the entities of that instance of the host subprogram.

All other entities, including saved data objects, are common to all instances of the subprogram. For example, the value of a saved data object appearing in one instance may have been defined in a previous instance or by an INITIAL attribute or DATA statement.

12.5.2.5 ENTRY Statement.

R1224 *entry-stmt* **is** ENTRY *entry-name* [([*dummy-arg-list*])]

Constraint: A *dummy-arg* may be an alternate return indicator only if the ENTRY statement is contained in a subroutine subprogram.

If the ENTRY statement is contained in a function subprogram, an additional function is defined by that subprogram. The name of the function and its result variable is *entry-name*. The characteristics of the function result are specified by specifications of *entry-name*. The dummy arguments of the function are those specified on the ENTRY statement. If the characteristics of the result of the function named on the ENTRY statement are the same as the characteristics of the function named on the FUNCTION statement, their result variables are associated. Otherwise, they are storage associated with the restrictions that they are scalar, that they have type and type parameters permitting storage association, and that they have the same lengths if they are of character type,

If the ENTRY statement is contained in a subroutine subprogram, an additional subroutine is defined by that subprogram. The name of the subroutine is *entry-name*. The dummy arguments of the subroutine are those specified on the ENTRY statement.

12.5.2.6 RETURN Statement.

R1225 *return-stmt* **is** RETURN [*scalar-int-expr*]

Constraint: The *return-stmt* must be contained in a function or subroutine subprogram.

Constraint: The *scalar-int-expr* is allowed only in a subroutine subprogram.

Constraint: The *expression* must produce a scalar result of type integer.

Execution of the RETURN statement completes execution of the instance of the subprogram in which it appears. If the expression is present and has a value *n* between 1 and the number of asterisks in the dummy argument list, the CALL statement that invoked the subroutine transfers control to the statement identified by the *n*th alternate return specifier in the actual argument list. If the expression is omitted or has a value outside the required range, there is no transfer of control to an alternate return.

Execution of an END statement, an END FUNCTION statement, or END SUBROUTINE statement is equivalent to executing a RETURN statement with no expression.

12.5.2.7 CONTAINS Statement. The CONTAINS statement separates the body of a program unit from any internal subprograms it may contain. Execution of the CONTAINS statement in a main program or procedure subprogram causes transfer of control to the END PROGRAM, END FUNCTION, or END SUBROUTINE statement of the program in which it appears. A CONTAINS statement in a module subprogram is not executable.

12.5.2.8 Restrictions on Dummy Arguments Not Associated. A dummy argument in an instance of a procedure subprogram must be associated with an actual argument unless it is optional and no corresponding actual argument is supplied when the procedure is invoked or unless the subprogram defines multiple procedures and the dummy argument is not part of the argument list of the procedure invoked to create this instance of the subprogram. A dummy argument not associated with an actual argument is subject to the following restrictions:

(1) If it is a dummy data object, it must not be referenced or be defined.

(2) If it is a dummy procedure, it must not be invoked.

(3) If it is a dummy condition, it must not be signaled.

(4) It must not be supplied as an actual argument corresponding to a nonoptional dummy argument other than the argument of the PRESENT intrinsic function.

(5) It may be supplied as an actual argument corresponding to an optional dummy argument. The optional dummy argument is then also considered not to be associated with an actual argument.

12.5.2.9 Restrictions on Entities Associated with Dummy Arguments. While an entity is associated with a dummy argument, the following restrictions hold:

(1) No action may be taken that affects the availability of that entity. Note that an allocatable entity may not be deallocated. For example, the internal procedure

```
SUBROUTINE INNER (A)
   USE ONLY (B)
   REAL A (:)
   DEALLOCATE (B)
   ...
```

must not be invoked from its host by the statement

```
CALL INNER (B)
```

(2) If the dummy argument does not have intent IN, no action may be taken that defines any part of that entity unless it does so through the dummy argument. For example, if a subroutine is headed by

```
SUBROUTINE SUB (A, B)
```

and is invoked by

```
CALL SUB (C, C)
```

(3) If any part of the entity is defined through the dummy argument, it may be referenced only through that dummy argument.

A must not be defined during execution of SUB, because in this case B would be defined by a means other than through its own argument association.

(4) If the dummy argument has intent IN, no reference to any part of it may be made after the definition status of the corresponding part of the actual argument has been changed. For example, the following reference

```
USE MODL, ONLY (D)
CALL SUB (D)
```

to a subroutine SUB of the form:

```
SUBROUTINE SUB (A)
   USE MODL, ONLY (C, D)
   REAL, INTENT (IN) :: A
   D = 1.0
   C = A * B
END
```

is not permitted because the value of the dummy argument A is used in a reference after the associated object D is defined through its own name.

12.5.3 Definition of Procedures by Means Other Than Fortran. The means other than Fortran by which a procedure may be defined are processor dependent. A reference to such a procedure is made as though it were defined by a procedure subprogram. The definition of a non-Fortran procedure must not be contained in a Fortran program unit and a Fortran program unit must not be contained in the definition of a non-Fortran procedure.

12.5.3.1 Statement Function.

R1226 *stmt-function-stmt* **is** *function-name* ([*dummy-arg-name-list*]) = *expr*

Constraint: The *expr* may be composed only of constants (literal and symbolic), references to scalar variables and array elements, references to functions, and intrinsic operators. If a reference to another statement function appears in *expr*, its definition must have been provided earlier in the program unit.

Constraint: The *function-name* and each *dummy-arg-spec* must be specified, explicitly or implicitly, to be scalar data objects.

The statement function produces the same result value as an internal function of the form

```
FUNCTION function-name ([dummy-arg-name-list])
  function-and-dummy-specifications
  function-name = expr
END FUNCTION function-name
```

where *function-and-dummy-specifications* are the specifications necessary to cause *function-name* and each *dummy-arg-spec* to be given explicitly the same type and type parameters that those names are given, explicitly or implicitly, in the program unit containing the statement function. Note, however, that unlike the internal function, the statement function always has an implicit interface and may not be supplied as a procedure argument.

12.5.4 Overloading Names. Two or more functions may be accessible with the same name in the same program scope. Similarly, two or more functions may be accessible with the same operator symbol in the same program scope; two or more subroutines may be accessible with the same name in the same program scope; and two or more subroutines may be accessible as assignments in the same program scope (Section 14).

13 INTRINSIC PROCEDURES

13.1 Intrinsic Functions.

An **intrinsic function** is either an inquiry function, an elemental function, or a transformational function. An **inquiry function** is one whose result depends on the explicit or implicit declarations associated with its principal argument and not on the value of this argument; in fact, the argument value may be undefined. An **elemental function** is one that is specified for scalar arguments, but may be applied to array arguments as described in 13.2. All other intrinsic functions are **transformational functions**; they almost all have one or more array-valued arguments or an array-valued result.

Generic names of intrinsic functions are listed in 13.8.1 through 13.8.14. In most cases, generic functions accept arguments of more than one type and the type of the result is the same as the type of the arguments. **Specific names** of intrinsic functions with corresponding generic names are listed in 13.8.17.

If an intrinsic function is used as an actual argument to an external procedure, its specific name must be used and it may be referenced in the external procedure only with scalar arguments. If an intrinsic function does not have a specific name, it must not be used as an actual argument.

13.2 Elemental 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 rank. If the arguments are all scalar, the result is scalar. For those elemental intrinsic functions that have more than one argument, all arguments must be conformable. In the array-valued case, the values of the elements of the result are the same as would have been obtained if the scalar-valued function had been applied separately to corresponding elements of each argument.

13.3 Argument Presence and Condition Status Functions.

The inquiry function PRESENT permits an inquiry to be made about the presence of an actual argument associated with a dummy argument. The inquiry functions ENABLED and HANDLED permit inquiries to be made about whether a condition has been enabled or would be handled.

13.4 Numeric, Mathematical, Bit, Character, and Derived-Type Functions.

13.4.1 Numeric Functions. The elemental functions INT, REAL, DBLE, and CMPLX perform type conversions. The elemental functions AIMAG, CONJG, AINT, ANINT, NINT, ABS, MOD, SIGN, DIM, DPROD, MAX, and MIN perform simple numeric operations.

13.4.2 Mathematical Functions. The elemental functions SQRT, EXP, LOG, LOG10, SIN, COS, TAN, ASIN, ACOS, ATAN, ATAN2, SINH, COSH, and TANH evaluate elementary mathematical functions.

13.4.3 Bit Functions. The elemental functions LBIT and BITL convert between bit and logical type. The transformational functions IBITLR and BITLR convert between a bit array and an integer, counting bits from left to right; IBITRL and BITRL are similar functions that count bits from right to left.

13.4.4 Bit Inquiry Functions. The inquiry function MAXBITS returns the maximum size of a bit array that can be converted to an integer.

13.4.5 Character Functions. The elemental functions ICHAR, CHAR, LGE, LGT, LLE, LLT, IACHAR, ACHAR, INDEX, ADJUSTL, ADJUSTR, REPEAT, ISCAN, and LEN_TRIM perform character operations. The TRIM function returns the argument with trailing blanks removed.

13.4.6 CHARACTER Inquiry Functions. The inquiry function LEN returns the length of a character entity.

13.4.7 Derived Data Type Inquiry Functions. A derived data type definition that includes a dummy type parameter list causes the implicit definition of a set of inquiry functions, one for each type parameter. These inquiry functions have names which are the same as the dummy parameter names. Each has a single argument whose type must be that defined by the type definition and returns a single integer result. The result is the value of the indicated parameter for the structure that is the argument.

The scope of these implicitly defined inquiry functions is the same as that of the derived data type. These functions may be referenced in any program unit in which the derived data type definition may be referenced. Note that the argument need not be defined at the time the function is referenced. For example, if

```
TYPE (STRING (100)) :: LINE
```

declares an object of the type STRING as defined in 4.4.1.1, the function reference MAX_SIZE (LINE) returns the integer result 100.

13.5 Numeric Manipulation and Inquiry Functions.

The floating point manipulation and inquiry functions are described in terms of a model for the representation and behavior of real numbers on a processor. The model has parameters which are determined so as to make the model best fit the machine on which the executable program is executed.

13.5.1 Models for Integer and Real Data. The model set for integer *i* is defined by:

$$i = s \times \sum_{k=1}^{q} w_k \times r^{k-1}$$

where r is an integer exceeding one, q is a positive integer, each w_k is a nonnegative integer less than r, and s is +1 or -1. The model set for real x is defined by:

$$x = \begin{cases} 0 \ \ or \\ s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k}, \end{cases}$$

where b and p are integers exceeding one; each f_k is a nonnegative integer less than b, except f_1 which is also nonzero; s is +1 or −1; and e is an integer that lies between some integer maximum $e_{\max}$ and some integer minimum $e_{\min}$ inclusively. The integer parameters r and q determine the set of model integers and the integer parameters b, p, $e_{\min}$, and $e_{\max}$ determine the set of model floating point numbers. The parameters of the integer and real models are available for each integer and the real data type implemented by the processor. The parameters characterize the set of available numbers in the definition of the model. The floating point manipulation and inquiry functions provide values related to the parameters and other constants related to them. For examples of the use of these functions, use the models:

$$i = s \times \sum_{k=1}^{31} w_k \times 2^{k-1}$$

and

$$x = s \times 2^e \times \left(½ + \sum_{k=2}^{24} f_k \times 2^{-k} \right), \quad -126 \le e \le 127$$

13.5.2 Numeric Inquiry Functions. The inquiry functions RADIX, DIGITS, MINEXP, MAXEXP, HUGE, TINY, EPSILON, EFFECTIVE_PRECISION, and EFFECTIVE_-EXPONENT_RANGE return scalar values related to the parameters of the model associated with the type and type parameters of the arguments. The value of the arguments to these functions need not be defined.

It is not necessary for a processor to evaluate the arguments of a numeric inquiry function if the value of the function can be determined otherwise.

13.5.3 Floating Point Manipulation Functions. The elemental functions EXPONENT, SCALE, NEAREST, FRACTION, SETEXPONENT, SPACING, and RRSPACING return values related to the components of the model values (13.5.1) associated with the actual values of the arguments.

13.6 Array Intrinsic Functions.

The array intrinsic functions perform the following operations on arrays: vector and matrix multiplication, numeric or logical computation that reduces the rank, array structure inquiry, array construction, array manipulation, and geometric location.

13.6.1 The Shape of Array Arguments. The inquiry and transformational array intrinsic functions operate on each array argument as a whole. The declared shape or effective shape of the corresponding actual argument must therefore be defined; that is, the actual argument must be an array section, an assumed-shape array, an explicit-shape array, an allocatable array that has been allocated, an alias array that exists, or an array-valued expression. It must not be an assumed-size array.

Some of the inquiry intrinsic functions accept array arguments for which the shape need not be defined. Assumed-size arrays may be used as arguments to these functions; they include the numeric inquiry functions, the functions RANK, ELBOUND, and DLBOUND, and certain references to SIZE, EUBOUND, and DUBOUND.

13.6.2 Mask Arguments. Some array intrinsic functions have an optional MASK argument that is used by the function to select the elements of one or more arguments to be operated on by the function. Any element not selected by the mask need not be defined at tht eime the function is invoked.

The MASK affects only the value of the function, and does not affect the evaluation, prior to invoking the function, of arguments that are array expressions.

A MASK argument may be of type LOGICAL or BIT. When the argument is of type BIT, a B'1' value is interpreted as .TRUE. and a B'0' is interpreted as .FALSE.

13.6.3 Vector and Matrix Multiplication Functions. The matrix multiplication function MATMUL operates on two matrices, or on one matrix and one vector, and returns the corresponding matrix-matrix, matrix-vector, or vector-matrix product. The arguments to MATMUL are arrays of the same type, which may be numeric (integer, real, double precision, or complex) or logical. On logical matrices and vectors, MATMUL performs Boolean multiplication.

The dot product function DOTPRODUCT operates on two vectors and returns their scalar product. The vectors are of the same type (numeric or logical) as for MATMUL. For logical vectors, DOTPRODUCT returns the Boolean scalar product.

13.6.4 Array Reduction Functions. The array reduction functions SUM, PRODUCT, MAXVAL, MINVAL, COUNT, ANY, and ALL perform numerical, logical, and counting operations on arrays. They may be applied to the whole array to give a scalar result or they may be applied over a given dimension to yield a result of rank reduced by one. By use of a logical mask that is conformable with the given array, the computation may be confined to any subset of the array (e.g., the positive elements).

13.6.5 Array Inquiry Functions. The array inquiry function RANK returns the number of dimensions of its argument. The functions SIZE, SHAPE, ELBOUND, and EUBOUND return, respectively, the effective number of elements, the effective sizes along each dimensions, and the effective lower and upper bounds of the subscripts along each dimension. The functions DSIZE, DSHAPE, DLBOUND, and DUBOUND return, respectively, the declared size of the array, the declared shape, and the declared lower and upper bounds of the subscripts along each dimension.

The values of the array arguments to these functions need not be defined.

It is not necessary for a processor to evaluate the arguments of an array inquiry function if the value of the function can be determined otherwise.

13.6.6 Array Construction Functions. The functions MERGE, SPREAD, REPLICATE, RESHAPE, DIAGONAL, PACK, and UNPACK construct new arrays from the elements of existing ones. MERGE combines two conformable arrays into one by an element-wise choice based on a logical mask. SPREAD and REPLICATE construct an array from several copies of an actual argument (SPREAD does this by adding an extra dimension, as in forming a book from copies of one page; REPLICATE does it by increasing the size of one of the dimensions as in laying the pages side by side). RESHAPE produces an array with the same elements and a different shape. DIAGONAL constructs a diagonal matrix. PACK and UNPACK respectively gather and scatter the elements of a one-dimensional array from and to positions in another array where the positions are specified by a logical mask.

13.6.7 Array Manipulation Functions. The functions TRANSPOSE, EOSHIFT, and CSHIFT manipulate arrays. TRANSPOSE performs the matrix transpose operation on a two-dimensional array. The shift functions leave the shape of an array unaltered but shift the positions of the elements parallel to a specified dimension of the array. These shifts are either circular (CSHIFT), in which case elements shifted off one end reappear at the other end, or end-off (EOSHIFT), in which case specified boundary elements are shifted into the vacated positions.

13.6.8 Array Geometric Location Functions. The geometric location functions FIRSTLOC, LASTLOC, and PROJECT provide access to the boundaries, or edges, of any subset of an array defined by a logical mask. FIRSTLOC and LASTLOC operate on the mask to define the edges of the subset and PROJECT extracts the elements that lie along an edge. For example, to extract from the integer table TABLE (M,N) the vector containing the first positive number in each column, first locate the desired elements in a logical mask FST (M,N) by:

```
FST = FIRSTLOC (TABLE .GT. 0, DIM = 1)
```

and then assign the elements to FSTC by:

```
FSTC = PROJECT (TABLE, FST, DIM = 1, FIELD = 0)
```

The functions MAXLOC and MINLOC return the location (subscripts) of an element of an array that has maximum and minimum values, respectively. By use of an optional logical mask that is conformable with the given array, the reduction may be confined to any subset of the array.

13.7 Intrinsic Subroutines.

Intrinsic subroutines are supplied by the processor and have the special definitions given in 13.9. An intrinsic subroutine is referenced by a CALL statement that uses its name explicitly. The name of an intrinsic subroutine must not be used as an actual argument. The effect of a subroutine reference is as specified in 13.9.

13.7.1 Date and Time Subroutines. The subroutines DATE_AND_TIME and CLOCK return integer data from the date and real-time clock. The time returned is local, but there are facilities for finding out the difference between local time and Greenwich Mean Time.

13.8 Tables of Generic Intrinsic Functions.

13.8.1 Argument Presence and Condition Status Functions.

ENABLED (CONDITION, LEVEL) Optional LEVEL	Condition enabled
HANDLED (CONDITION, LEVEL) Optional LEVEL	Condition handled
PRESENT (A)	Argument presence

13.8.2 Numeric Functions.

ABS (A)	Absolute value
AIMAG (Z)	Imaginary part of a complex number
AINT (A)	Truncation to whole number
ANINT (A)	Nearest whole number
CMPLX (X, Y, MOLD) Optional Y, MOLD	Conversion to complex type
CONJG (Z)	Conjugate of a complex number
DBLE (A)	Conversion to double precision type
DIM (X, Y)	Positive difference
DPROD (X, Y)	Double precision product
INT (A)	Conversion to integer type
MAX (A1, A2, A3,...) Optional A3,...	Maximum value
MIN (A1, A2, A3,...) Optional A3,...	Minimum value
MOD (A, P)	Remainder modulo P
NINT (A)	Nearest integer
REAL (A, MOLD) Optional MOLD	Conversion to real type
SIGN (A, B)	Transfer of sign

13.8.3 Mathematical Functions.

ACOS (X)	Arccosine
ASIN (X)	Arcsine
ATAN (X)	Arctangent
ATAN2 (Y, X)	Arctangent
COS (X)	Cosine

COSH (X)	Hyperbolic cosine
EXP (X)	Exponential
LOG (X)	Natural logarithm
LOG10 (X)	Common logarithm (base 10)
SIN (X)	Sine
SINH (X)	Hyperbolic sine
SQRT (X)	Square root
TAN (X)	Tangent
TANH (X)	Hyperbolic tangent

13.8.4 Bit Functions.

BITL (L)	Convert from logical to bit type
BITLR (I,SIZE) Optional SIZE	Convert an integer to a bit array, counting left to right
BITRL (I,SIZE) Optional SIZE	Convert an integer to a bit array, counting right to left
IBITLR (B)	Convert a bit array to an integer, counting left to right
IBITRL (B)	Convert a bit array to an integer, counting right to left
LBIT (B)	Convert from bit to logical type

13.8.5 Bit Inquiry Functions.

MAXBITS (I)	Maximum bit array length for conversion

13.8.6 Character Functions.

ACHAR (I)	Character in given position in ASCII collating sequence
ADJUSTL (STRING)	Adjust left
ADJUSTR (STRING)	Adjust right
CHAR (I)	Character in given position in processor collating sequence
IACHAR (C)	Position of a character in ASCII collating sequence
ICHAR (C)	Position of a character in processor collating sequence
INDEX (STRING, SUBSTRING)	Starting position of a substring
ISCAN (STRING, SET)	Scan a string for a character in a set
LEN_TRIM (STRING)	Length without trailing blank characters
LGE (STRING_A, STRING_B)	Lexically greater than or equal
LGT (STRING_A, STRING_B)	Lexically greater than
LLE (STRING_A, STRING_B)	Lexically less than or equal
LLT (STRING_A, STRING_B)	Lexically less than
REPEAT (STRING, NCOPIES)	Repeated concatenation
TRIM (STRING)	Remove trailing blank characters
VERIFY (STRING, SET)	Verify the set of characters in a string

13.8.7 Character Inquiry Functions.

LEN (STRING)	Length of a character entity

13.8.8 Numeric Inquiry Functions.

DIGITS (X)	Number of significant digits in the model
EFFECTIVE_EXPONENT_RANGE (X)	Effective decimal exponent range
EFFECTIVE_PRECISION (X)	Effective decimal precision
EPSILON (X)	Number that is almost negligible compared to one
HUGE (X)	Largest number in the model
MAXEXPONENT (X)	Maximum exponent in the model
MINEXPONENT (X)	Minimum exponent in the model
RADIX (X)	Base of the model
TINY (X)	Smallest number in the model

13.8.9 Floating-point Manipulation Functions.

EXPONENT (X)	Exponent part of a model number
FRACTION (X)	Fractional part of a number
NEAREST (X, S)	Nearest different processor number in given direction
RRSPACING (X)	Reciprocal of the relative spacing of model numbers near given number
SCALE (X, I)	Multiply a real by its base to an integer power
SETEXPONENT (X, I)	Set exponent part of a number
SPACING (X)	Absolute spacing of model numbers near given number

13.8.10 Vector and Matrix Multiply Functions.

DOTPRODUCT (VECTOR_A, VECTOR_B)	Dot product of two arrays
MATMUL (MATRIX_A, MATRIX_B)	Matrix multiplication

13.8.11 Array Reduction Functions.

ALL (MASK, DIM) Optional DIM	True if all values are true
ANY (MASK, DIM) Optional DIM	True if any value is true
COUNT (MASK, DIM) Optional DIM	Number of true elements in an array
MAXVAL (ARRAY, DIM, MASK) Optional DIM, MASK	Maximum value in an array
MINVAL (ARRAY, DIM, MASK) Optional DIM, MASK	Minimum value in an array
PRODUCT (ARRAY, DIM, MASK) Optional DIM, MASK	Product of array elements
SUM (ARRAY, DIM, MASK) Optional DIM, MASK	Sum of array elements

13.8.12 Array Inquiry Functions.

ALLOCATED (ARRAY)	Array allocation
DLBOUND (ARRAY, DIM) Optional DIM	Declared lower dimension bounds of an array
DUBOUND (ARRAY, DIM) Optional DIM	Declared upper dimension bounds of an array

DSHAPE (SOURCE)	Declared shape of an array or scalar
DSIZE (ARRAY, DIM) Optional DIM	Total number of elements in declared array
EUBOUND (ARRAY, DIM) Optional DIM	Effective upper dimension bounds of an array
ELBOUND (ARRAY, DIM) Optional DIM	Effective lower dimension bounds of an array
ESHAPE (SOURCE)	Effective shape of an array or scalar
ESIZE (ARRAY, DIM) Optional DIM	Total number of elements in effective array
RANK (SOURCE)	Rank of an array or scalar

13.8.13 Array Construction Functions.

DIAGONAL (ARRAY, FILL) Optional FILL	Create a diagonal matrix
MERGE (TSOURCE, FSOURCE, MASK)	Merge under mask
PACK (ARRAY, MASK, VECTOR) Optional VECTOR	Pack an array into an array of rank one under a mask
RESHAPE (MOLD, SOURCE, PAD, ORDER) Optional PAD, ORDER	Reshape an array
REPLICATE (ARRAY, DIM, NCOPIES)	Replicates array by increasing a dimension
SPREAD (SOURCE, DIM, NCOPIES)	Replicates array by adding a dimension
UNPACK (VECTOR, MASK, FIELD)	Unpack an array of rank one into an array under a mask

13.8.14 Array Manipulation Functions.

CSHIFT (ARRAY, DIM, SHIFT)	Circular shift
EOSHIFT (ARRAY, DIM, SHIFT, BOUNDARY) Optional BOUNDARY	End-off shift
TRANSPOSE (MATRIX)	Transpose of an array of rank two

13.8.15 Array Geometric Location Functions.

FIRSTLOC (MASK, DIM) Optional DIM	Locate first true element
LASTLOC (MASK, DIM) Optional DIM	Locate last true element
MAXLOC(ARRAY,MASK) Optional MASK	Location of a maximum value in an array
MINLOC(ARRAY,MASK) Optional MASK	Location of a minimum value in an array
PROJECT (ARRAY, MASK, FIELD, DIM) Optional DIM	Select masked values

13.8.16 Table of Intrinsic Subroutines.

CLOCK (COUNT, COUNT_RATE, COUNT_MAX) Optional COUNT, COUNT_RATE, COUNT_MAX	Obtain data from the system clock
DATE_AND_TIME (ALL, COUNT, MSECOND, SECOND, MINUTE, HOUR, DAY, MONTH, YEAR, ZONE) Optional ALL, COUNT, MSECOND, SECOND, MINUTE, HOUR, DAY, MONTH, YEAR, ZONE	Obtain date and time

13.8.17 Table of Specific Intrinsic Functions.

	Specific Name	*Generic Name*	*Argument Type*
	ABS(A)	ABS(A)	real
	ACOS(X)	ACOS(X)	real
	AIMAG(Z)	AIMAG(Z)	complex
	AINT(A)	AINT(A)	real
	ALOG(X)	LOG(X)	real
	ALOG10(X)	LOG10(X)	real
•	AMAX0(A1,A2,A3,...) Optional A3,...	REAL(MAX(A1, A2,A3,...)) Optional A3,...	integer
•	AMAX1(A1,A2,A3,...) Optional A3,...	MAX(A1, A2,A3,...) Optional A3,...	real
•	AMIN0(A1,A2,A3,...) Optional A3,...	REAL(MIN(A1, A2,A3,...)) Optional A3,...	integer
•	AMIN1(A1,A2,A3,...) Optional A3,...	MIN(A1, A2,A3,...) Optional A3,...	real
	AMOD(A,P)	MOD(A,P)	real
	ANINT(A)	ANINT(A)	real
	ASIN(X)	ASIN(X)	real
	ATAN(X)	ATAN(X)	real
	ATAN2(Y,X)	ATAN2(Y,X)	real
	CABS(A)	ABS(A)	complex
	CCOS(X)	COS(X)	complex
	CEXP(X)	EXP(X)	complex
•	CHAR(I)	CHAR(I)	integer
	CLOG(X)	LOG(X)	complex
	CONJG(Z)	CONJG(Z)	complex
	COS(X)	COS(X)	real
	COSH(X)	COSH(X)	real
	CSIN(X)	SIN(X)	complex
	CSQRT(X)	SQRT(X)	complex
	DABS(A)	ABS(A)	double precision
	DACOS(X)	ACOS(X)	double precision
	DASIN(X)	ASIN(X)	double precision

	DATAN(X)	ATAN(X)	double precision
	DATAN2(Y,X)	ATAN2(Y,X)	double precision
	DCOS(X)	COS(X)	double precision
	DCOSH(X)	COSH(X)	double precision
	DDIM(X,Y)	DIM(X,Y)	double precision
	DEXP(X)	EXP(X)	double precision
	DIM(X,Y)	DIM(X,Y)	real
	DINT(A)	AINT(A)	double precision
	DLOG(X)	LOG(X)	double precision
	DLOG10(X)	LOG10(X)	double precision
•	DMAX1(A1,A2,A3,...)	MAX(A1,A2,A3,...)	double precision
	Optional A3,...	Optional A3,...	
•	DMIN1(A1,A2,A3,...)	MIN(A1,A2,A3,...)	double precision
	Optional A3,...	Optional A3,...	
	DMOD(A,P)	MOD(A,P)	double precision
	DNINT(A)	ANINT(A)	double precision
	DPROD(X,Y)	DPROD(X,Y)	real
	DSIN(A,B)	SIGN(A,B)	double precision
	DSIGN(X)	SIN(X)	double precision
	DSINH(X)	SINH(X)	double precision
	DSQRT(X)	SQRT(X)	double precision
	DTAN(X)	TAN(X)	double precision
	DTANH(X)	TANH(X)	double precision
	EXP(X)	EXP(X)	real
•	FLOAT(A)	REAL(A)	integer
	IABS(A)	ABS(A)	integer
•	ICHAR(C)	ICHAR(C)	character
	IDIM(X,Y)	DIM(X,Y)	integer
•	IDINT(A)	INT(A)	double precision
	IDNINT(A)	NINT(A)	double precision
•	IFIX(A)	INT(A)	real
	INDEX(S,T)	INDEX(S,T)	character
•	INT(A)	INT(A)	real
	ISIGN(A,B)	SIGN(A,B)	integer
	LEN(S)	LEN(S)	character
•	LGE(S,T)	LGE(S,T)	character
•	LGT(S,T)	LGT(S,T)	character
•	LLE(S,T)	LLE(S,T)	character
•	LLT(S,T)	LLT(S,T)	character
•	MAX0(A1,A2,A3,...)	MAX(A1,A2,A3,...)	integer
	Optional A3,...	Optional A3,...	
•	MAX1(A1,A2,A3,...)	INT(MAX(A1,A2,A3,...))	real
	Optional A3,...	Optional A3,...	
•	MIN0(A1,A2,A3,...)	MIN(A1,A2,A3,...)	integer
	Optional A3,...	Optional A3,...	
•	MIN1(A1,A2,A3,...)	INT(MIN(A1,A2,A3,...))	real
	Optional A3,...	Optional A3,...	
	MOD(A,P)	MOD(A,P)	integer
	NINT(A)	NINT(A)	real
•	REAL(A)	REAL(A)	integer
	SIGN(A,B)	SIGN(A,B)	real
	SIN(X)	SIN(X)	real
	SINH(X)	SINH(X)	real
•	SNGL(A)	REAL(A)	double precision

SQRT(X)	SQRT(X)	real
TAN(X)	TAN(X)	real
TANH(X)	TANH(X)	real

• These specific intrinsic function names must not be used as an actual argument.

13.9 Specifications of the Intrinsic Procedures.

This section contains detailed specifications of all the intrinsic procedures.

13.9.1 ABS (A).

Description. Absolute value.

Kind. Elemental function.

Argument. A must be of type integer, real, double precision, or complex.

Result Type and Type Parameters. The same as A except that if A is complex, the result is real.

Result Value. If A is of type integer, real, or double precision, the value of the result is $|A|$; if A is complex with value (x,y), the result is equal to a processor-dependent approximation to $\sqrt{x^2+y^2}$.

Example. ABS ((3.0, 4.0)) has the value 5.0 (approximately).

13.9.2 ACHAR (I).

Description. Returns the character in a specified position of the ASCII collating sequence. It is the inverse of the IACHAR function.

Kind. Elemental function.

Argument. I must be of type integer.

Result Type and Type Parameters. Character of length one.

Result Value. If I has value in the range $0 \le I \le 127$, the result is the character in position I of the ASCII collating sequence; otherwise, the result is processor dependent. If the processor is not capable of representing both upper and lower case letters and I corresponds to an ASCII letter in a case that the processor is not capable of representing, the result is the letter in the case that the processor is capable of representing. ACHAR (IACHAR (C)) must have the value C for any character C capable of representation in the processor.

Example. ACHAR (88) has the value 'X'.

13.9.3 ACOS (X).

Description. Arccosine (inverse cosine) function.

Kind. Elemental function.

Argument. X must be of type real or double precision with a value that satisfies the inequality $|X| \le 1$.

Result Type and Type Parameters. Same as X.

Result Value. The result has value equal to a processor-dependent approximation to arccos(X), expressed in radians. It lies in the range $0 \le \text{ACOS (X)} \le \pi$.

Example. ACOS (0.54030231) has the value 1.0 (approximately).

13.9.4 ADJUSTL (STRING).

Description. Adjust to the left, removing leading blanks and inserting trailing blanks.

Kind. Elemental function.

Argument. STRING must be of type character.

Result Type and Type Parameters. Character of the same length as STRING.

Result Value. The value of the result is the same as STRING except that any leading blanks have been deleted and the same number of trailing blanks have been inserted.

Example. ADJUSTL ('   WORD') has value 'WORD   '.

13.9.5 ADJUSTR (STRING).

Description. Adjust to the right, removing trailing blanks and inserting leading blanks.

Kind. Elemental function.

Argument. STRING must be of type character.

Result Type and Type Parameters. Character of the same length as STRING.

Result Value. The value of the result is the same as STRING except that any trailing blanks have been deleted and the same number of leading blanks have been inserted.

Example. ADJUSTR ('WORD   ') has value '   WORD'.

13.9.6 AIMAG (Z).

Description. Imaginary part of a complex number.

Kind. Elemental function.

Argument. Z must be of type complex.

Result Type and Type Parameters. Real with the same type parameters as Z.

Result Value. If Z has the value (x, y), the result has value y.

Example. AIMAG ((2.0, 3.0)) has the value 3.0.

13.9.7 AINT (A).

Description. Truncation to a whole number.

Kind. Elemental function.

Argument. A must be of type real or double precision.

Result Type and Type Parameters. Same as A.

Result Value. If $|A| < 1$, AINT (A) has the value 0; if $|A| \geq 1$, AINT (A) has value equal to the largest integer that does not exceed the magnitude of A and whose sign is the same as the sign of A.

Example. AINT (2.783) has the value 2.0.

13.9.8 ALL (MASK, DIM).

Optional Argument. DIM

Description. Determine whether all values are true in ARRAY along dimension DIM.

Kind. Transformational function.

Arguments.

ARRAY	must be of type logical or bit. It must not be scalar.
DIM (optional)	must be scalar and of type integer with value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY.

Result Type and Shape. The result is of type logical. It is scalar if DIM is absent or ARRAY has rank one; otherwise, the result is an array of rank $n-1$ and of shape (d_1, d_2, ..., $d_{\text{DIM}-1}$, $d_{\text{DIM}+1}$, ..., d_n) where (d_1, d_2, ..., d_n) is the shape of ARRAY.

Result Value.

Case (i): The result of ALL (MASK) has value .TRUE. if all elements of ARRAY are true or if ARRAY has size zero, and the result has value .FALSE. if any element of ARRAY is false.

Case (ii): If ARRAY has rank one, ALL (MASK, DIM) has value equal to that of ALL (MASK). Otherwise, the value of element (s_1, s_2, ..., $s_{\text{DIM}-1}$, $s_{\text{DIM}+1}$, ..., s_n) of ALL (MASK, DIM) is equal to ALL (MASK (s_1, s_2, ..., $s_{\text{DIM}-1}$, :, $s_{\text{DIM}+1}$, ..., s_n)).

Examples.

Case (i): The value of ALL ([.TRUE., .FALSE., .TRUE.]) is .FALSE.

Case (ii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ and C is the array $\begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$, then ALL (B .NE. C, DIM = 1) is [.TRUE., .FALSE., .FALSE.] and ALL (B .NE. C, DIM = 2) is [.FALSE., .FALSE.].

13.9.9 ALLOCATED (ARRAY).

Description. Indicate whether or not an allocatable array is currently allocated space.

Kind. Inquiry function.

Argument. ARRAY must be an allocatable array.

Result Type and Shape. The result is a logical scalar.

Result Value. The result has the value .TRUE. if ARRAY is currently allocated and has the value .FALSE. otherwise.

13.9.10 ANINT (A).

Description. Nearest whole number.

Kind. Elemental function.

Argument. A must be of type real or double precision.

Result Type and Type Parameters. Same as A.

Result Value. If $A > 0$, ANINT (A) has the value AINT (A + 0.5); if $A \leq 0$, ANINT (A) has the value AINT (A − 0.5).

Example. ANINT (2.783) has the value 3.0

13.9.11 ANY (MASK, DIM).

Optional Argument. DIM

Description. Determine whether any value is true in MASK along dimension DIM.

Kind. Transformational function.

Arguments.

MASK	must be of type logical or bit. It must not be scalar.
DIM (optional)	must be scalar and of type integer with value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of MASK.

Result Type and Shape. The result is of type logical. It is scalar if DIM is absent or MASK has rank one; otherwise, the result is an array of rank $n-1$ and of shape $(d_1, d_2, \ldots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \ldots, d_n)$ where $(d_1, d_2, \ldots, d_n)$ is the shape of MASK.

Result Value.

Case (i): The result of ANY (MASK) has value .TRUE. if any element of MASK is true and has value .FALSE. if no elements are true or if MASK has size zero.

Case (ii): If MASK has rank one, ANY (MASK, DIM) has value equal to that of ANY (MASK). Otherwise, the value of element $(s_1, s_2, \ldots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \ldots, s_n)$ of ANY (MASK, DIM) is equal to ANY (MASK $(s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)$).

Examples.

Case (i): The value of ANY ([.TRUE., .FALSE., .TRUE.]) is .TRUE.

Case (ii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ and C is the array $\begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$, ANY (B .NE. C, DIM = 1) is [.TRUE., .FALSE., .TRUE.] and ANY (B .NE. C, DIM = 2) is [.TRUE., .TRUE.].

13.9.12 ASIN (X).

Description. Arcsine (inverse sine) function.

Kind. Elemental function.

Argument. X must be of type real or double precision. Its value must satisfy the inequality $|X| \leq 1$.

Result Type and Type Parameters. Same as X.

Result Value. The result has value equal to a processor-dependent approximation to arcsin(X), expressed in radians. It lies in the range $-\pi/2 \leq \text{ASIN (X)} \leq \pi/2$.

Example. ASIN (0.84147098) has the value 1.0 (approximately).

13.9.13 ATAN (X).

Description. Arctangent (inverse tangent) function.

Kind. Elemental function.

Argument. X must be of type real or double precision.

Result Type and Type Parameters. Same as X.

Result Value. The result has the value equal to a processor-dependent approximation to arctan(X), expressed in radians, that lies in the range $-\pi/2 \leq$ ATAN (X) $\leq \pi/2$.

Example. ATAN (1.5574077) has the value 1.0 (approximately).

13.9.14 ATAN2 (Y, X).

Description. Arctangent (inverse tangent) function. The result is the principal value of the argument of the nonzero complex number (X, Y).

Kind. Elemental function.

Arguments.

Y	must be of type real or double precision.
X	must be of the same type as Y. If Y has value zero, X must not have value zero.

Result Type and Type Parameters. Same as X.

Result Value. The result has value equal to a processor-dependent approximation to the argument of the complex number (X, Y), expressed in radians. It lies in the range $-\pi <$ ATAN2 (Y, X) $\leq \pi$ and is equal to a processor-dependent approximation to a value of arctan(Y/X) if $X \neq 0$. If $Y > 0$, the result is positive. If $Y = 0$, the result is zero if $X > 0$ and the result is π if $X < 0$. If $Y < 0$, the result is negative. If $X = 0$, the absolute value of the result is $\pi/2$.

Example. ATAN2 (1.5574077, 1.0) has the value 1.0 (approximately).

13.9.15 BITL (L).

Description. Convert logical to bit type.

Kind. Elemental function.

Argument. L must be of type logical.

Result Type. Bit.

Result Value. The result has the value B'1' if L has the value .TRUE. and the value B'0' if L has the value .FALSE.

Example. BITL (.TRUE.) has the value B'1'.

13.9.16 BITLR (I, SIZE).

Optional Argument. SIZE

Description. Convert an integer to a bit array, counting left to right.

Kind. Transformational function.

Arguments.

I	must be scalar and of type integer. Its value must not be negative.
SIZE (optional)	must be scalar and of type integer with a positive value. If it is omitted, it is as if it were present with the value MAXBITS (1).

Result Type and Shape. The result is a bit array of rank one with SIZE number of elements.

Result Value. The result is a bit array containing the binary representation of the argument. The array element with the largest subscript value will contain the least significant bit of the binary representation. Zero extension or truncation will take place at the low end of the array as necessary. IBITLR (BITLR (J)) must have the value J for any value of the integer J. BITLR (IBITLR (B), SIZE (B)) must have the value B for any value of a bit array B for which SIZE (B) ≤ MAXBITS (1).

Example. BITLR (5, 6) has the value [B'0', B'0', B'0', B'1', B'0', B'1'].

13.9.17 BITRL (I, SIZE).

Optional Argument. SIZE

Description. Convert an integer to a bit array, counting right to left.

Kind. Transformational function.

Arguments.

I	must be scalar and of type integer. Its value must not be negative.
SIZE (optional)	must be scalar and of type integer with a positive value. If it is omitted, it is as if it were present with the value MAXBITS (1).

Result Type and Shape. The result is a bit array of rank one with SIZE number of elements.

Result Value. The result is a bit array containing the binary representation of the argument. The array element with the largest subscript value will contain the most significant bit of the binary representation. Zero extension or truncation will take place at the high end of the array as necessary. IBITRL (BITRL (J)) must have the value J for any value of the integer J. BITRL (IBITRL (B), SIZE (B)) must have the value B for any value of a bit array B for which SIZE (B) ≤ MAXBITS (1).

Example. BITRL(5,6) has the value [B'1', B'0', B'1', B'0', B'0', B'0'].

13.9.18 CHAR (I).

Description. Returns the character in a given position of the processor collating sequence. It is the inverse of the function ICHAR.

Kind. Elemental function.

Argument. I must be of type integer with value in the range $0 \le I \le n-1$, where n is the number of characters in the collating sequence.

Result Type and Type Parameters. Character of length one.

Result Value. The result is the character in position I of the processor collating sequence. ICHAR (CHAR (I)) must have the value I for $0 \le I \le n-1$ and CHAR (ICHAR (C)) must have the value C for any character C capable of representation in the processor.

Example. CHAR (88) has the value 'X' on a processor using the ASCII collating sequence.

13.9.19 CLOCK (COUNT, COUNT_RATE, COUNT_MAX).

Optional Arguments. COUNT, COUNT_RATE, COUNT_MAX

Description. Returns integer data from a real-time clock.

Kind. Subroutine.

Arguments.

COUNT (optional)	must be scalar and of type integer. It is set to a processor-dependent value based on the current value of the basic clock or to −HUGE (0) if there is no clock. The processor-dependent value is incremented by one for each clock count until the value COUNT_MAX is reached and is reset to zero at the next count. It lies in the range 0 to COUNT_MAX if there is a clock.
COUNT_RATE (optional)	must be scalar and of type integer. It is set to the number of basic clock counts per second, or to zero if there is no clock.
COUNT_MAX (optional)	must be scalar and of type integer. It is set to the maximum value that COUNT can have, or to zero if there is no clock.

Example. If the basic system clock is a 24-hour clock that registers time in 1-second intervals, at 11:30 A.M. the reference

```
CALL CLOCK (COUNT = C, COUNT_RATE = R, COUNT_MAX = M)
```

sets C = 11 × 3600 + 30 × 60 = 41,400 seconds, R = 1, and M = 24 × 2600 − 1 = 86,399 seconds.

13.9.20 CMPLX (X, Y, MOLD).

Optional Arguments. Y, MOLD

Description. Convert to complex type.

Kind. Elemental function.

Arguments.

X	must be of type integer, real, double precision, or complex.
Y (optional)	must be of type integer, real, or double precision. It must not be present if X is of type complex.
MOLD (optional)	must be of type real.

Result Type and Type Parameters. The result is of type complex. If MOLD is present, the type parameters are those of MOLD; otherwise, the type parameters are those of default real type.

Result Value. If Y is absent and X is not complex, it is as if Y were present with the value zero; if MOLD is absent, it is as if MOLD were present with default real type; CMPLX(X, Y, MOLD) has the complex value whose real part is REAL(X, MOLD) and whose imaginary part is REAL(Y, MOLD).

Example. CMPLX (−3) has the value (−3.0, 0.0).

13.9.21 CONJG (Z).

Description. Conjugate of a complex number.

Kind. Elemental function.

Argument. Z must be of type complex.

Result Type and Type Parameters. Same as Z.

Result Value. If Z has the value (x, y), the result has value $(x, -y)$.

Example. CONJG ((2.0, 3.0)) has the value (2.0, −3.0).

13.9.22 COS (X).

Description. Cosine function.

Kind. Elemental function.

Argument. X must be of type real, double precision, or complex.

Result Type and Type Parameters. Same as X.

Result Value. The result has value equal to a processor-dependent approximation to cos(X). If X is of type real or double precision, it is regarded as a value in radians. If X is of type complex, its real part is regarded as a value in radians.

Example. COS (1.0) has the value 0.54030231 (approximately).

13.9.23 COSH (X).

Description. Hyperbolic cosine function.

Kind. Elemental function.

Argument. X must be of type real or double precision.

Result Type and Type Parameters. Same as X.

Result Value. The result has value equal to a processor-dependent approximation to cosh(X).

Example. COSH (1.0) has the value 1.5430806 (approximately).

13.9.24 COUNT (MASK, DIM).

Optional Argument. DIM

Description. Count the number of true elements of MASK along dimension DIM.

Kind. Transformational function.

Arguments.

MASK	must be of type logical or bit. It must not be scalar.
DIM (optional)	must be scalar and of type integer with value in the range $1 \le \text{DIM} \le n$, where n is the rank of MASK.

Result Type and Shape. The result is of type integer. It is scalar if DIM is absent or MASK has rank one; otherwise, the result is an array of rank $n-1$ and of shape (d_1, d_2, ..., $d_{\text{DIM}-1}$, $d_{\text{DIM}+1}$, ..., d_n) where (d_1, d_2, ..., d_n) is the shape of MASK.

Result Value.

Case (i): The result of COUNT (MASK) has value equal to the number of true elements of MASK or has value zero if MASK has size zero.

Case (ii): If MASK has rank one, COUNT (MASK, DIM) has value equal to that of COUNT (MASK). Otherwise, the value of element (s_1, s_2, ..., $s_{\text{DIM}-1}$, $s_{\text{DIM}+1}$, ..., s_n) of COUNT (MASK, DIM) is equal to COUNT (MASK (s_1, s_2, ..., $s_{\text{DIM}-1}$, :, $s_{\text{DIM}+1}$, ..., s_n)).

Examples.

Case (i): The value of COUNT ([.TRUE., .FALSE., .TRUE.]) is 2.

Case (ii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ and C is the array $\begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$, COUNT (B .NE. C, DIM = 1) is [2, 0, 1] and COUNT (B .NE. C, DIM = 2) is [1, 2].

13.9.25 CSHIFT (ARRAY, DIM, SHIFT).

Description. Perform a circular shift on an array expression of rank one or perform circular shifts on all the complete rank one sections along a given dimension of a many-ranked array expression. Elements shifted out at one end of a section are shifted in at the other end. Different sections may be shifted by different amounts and in different directions.

Kind. Transformational function.

Arguments.

ARRAY	may be of any type. It must not be scalar.
DIM	must be a scalar and of type integer with value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY.
SHIFT	must be of type integer and must be scalar if ARRAY has rank one; otherwise, it must be scalar or of rank $n-1$ and of shape [E (1:DIM-1), E (DIM+1:n)] where E (1:n) is the shape of ARRAY.

Result Type, Type Parameters, and Shape. The result is of the type and type parameters of ARRAY, and has the shape of ARRAY.

Result Value.

Case (i): If ARRAY has rank one, the result is obtained by applying |SHIFT| circular shifts to ARRAY in the direction indicated by the sign of SHIFT. If SHIFT has value 1, element i of the result is ARRAY ($i+1$) for $i = 1, 2, ..., m-1$ and element m of the result is ARRAY (1) where m is the size of ARRAY. If SHIFT is positive, the result is equivalent to SHIFT applications of CSHIFT with SHIFT=1. If SHIFT has value -1, element i of the result is ARRAY ($i-1$) for $i = 2, 3, ..., m$ and element 1 of the result is ARRAY (m). If SHIFT is negative, the result is equivalent to $-$SHIFT applications of CSHIFT with SHIFT$= -1$.

Case (ii): If ARRAY has rank greater than one, section (s_1, s_2, ..., $s_{\text{DIM}-1}$, :, $s_{\text{DIM}+1}$, ..., s_n) of the result has value equal to CSHIFT (ARRAY (s_1, s_2, ..., $s_{\text{DIM}-1}$, :, $s_{\text{DIM}+1}$, ..., s_n, 1, sh), where sh is SHIFT or SHIFT (s_1, s_2, ..., $s_{\text{DIM}-1}$, $s_{\text{DIM}+1}$, ..., s_n).

Examples.

Case (i): If V is the array [1, 2, 3, 4, 5, 6], the effect of shifting V circularly to the left by two positions is achieved by CSHIFT (V, DIM=1, SHIFT=2) which has the value [3, 4, 5, 6, 1, 2]; CSHIFT (V, DIM=1, SHIFT=-2) achieves a circular shift to the right by two positions and has the value [5, 6, 1, 2, 3, 4].

Case (ii): The rows of an array of rank two may all be shifted by the same amount or by different amounts. If M is the array $\begin{bmatrix} A & B & C \\ A & B & C \\ A & B & C \end{bmatrix}$, the value of CSHIFT (M, DIM=2, SHIFT=-1) is $\begin{bmatrix} C & A & B \\ C & A & B \\ C & A & B \end{bmatrix}$, and the value of CSHIFT (M, DIM=2, SHIFT=[-1, 1, 0]) is $\begin{bmatrix} C & A & B \\ B & C & A \\ A & B & C \end{bmatrix}$.

13.9.26 DATE_AND_TIME (ALL, COUNT, MSECOND, SECOND, MINUTE, HOUR, DAY, MONTH, YEAR, ZONE).

Optional Arguments. ALL, COUNT, MSECOND, SECOND, MINUTE, HOUR, DAY, MONTH, YEAR, ZONE

Description. Returns integer data from the date available to the processor and a real-time clock.

Kind. Subroutine.

Arguments.

ALL (optional)	must be of type integer and rank one. Its size must be at least 9. The values returned in ALL are as for the remaining 9 arguments, taken in order.
COUNT (optional)	must be scalar and of type integer. It is set to a processor-dependent value based on the current value of the basic clock or to −HUGE (0) if there is no clock. The processor-dependent value is incremented by one for each clock count until the value COUNT_MAX (as returned by subroutine CLOCK) is reached and is reset to zero at the next count. It lies in the range 0 to COUNT_MAX if there is a clock.
MSECOND (optional)	must be scalar and of type integer. It is set to the millisecond part of the local time, or to −HUGE (0) if there is no clock. It lies in the range 0 to 999 if there is a clock.
SECOND (optional)	must be scalar and of type integer. It is set to the second part of the local time, or to −HUGE (0) if there is no clock. It lies in the range 0 to 59 if there is a clock.
MINUTE (optional)	must be scalar and of type integer. It is set to the minute part of the local time, or to −HUGE (0) if there is no clock. It lies in the range 0 to 59 if there is a clock.
HOUR (optional)	must be scalar and of type integer. It is set to the hour part of the local time, or to −HUGE (0) if there is no clock. It lies in the range 0 to 23 if there is a clock.
DAY (optional)	must be scalar and of type integer. It is set to the day of the month, or to −HUGE (0) if there is no date available. It lies in the range 1 to 31 if there is a date available.
MONTH (optional)	must be scalar and of type integer. It is set to the month of the year, or to −HUGE (0) if there is no date available. It lies in the range 1 to 12 if there is a date available.
YEAR (optional)	must be scalar and of type integer. It is set to the year according to the Gregorian calendar (e.g. 1988), or to −HUGE (0) if there is no date available.
ZONE (optional)	must be scalar and of type integer. It is set to the number of minutes that local time is behind Greenwich Mean Time, or to −HUGE (0) if there is no clock.

Example.

```
CALL DATE_AND_TIME (ZONE = HERE)
```

will assign the value 300 to the variable HERE if the local time is 5 hours behind GMT.

13.9.27 DBLE (A).

Description. Convert to double precision type.

Kind. Elemental function.

Argument. A must be of type integer, real, double precision, or complex.

Result Type. Double precision.

Result Value.

Case (i): If A is of type double precision, DBLE (A) = A.

Case (ii): If A is of type integer or real, the result is as much precision of the significant part of A as a double precision datum can contain.

Case (iii): If A is of type complex, the result is as much precision of the significant part of the real part of A as a double precision datum can contain.

Example. DBLE (−3) has the value −3.0D0.

13.9.28 DIAGONAL (ARRAY, FILL).

Optional Argument. FILL

Description. Create a diagonal matrix from its diagonal.

Kind. Transformational function.

Arguments.

ARRAY may be of any type. It must have rank one.

FILL (optional) must be of the same type and type parameters as ARRAY and must be scalar. It may be omitted for the data types in the following table; in this case it is as if it were present with the value shown.

Type of ARRAY	Value of FILL
Integer	0
Real	0.0
Double precision	0.0D0
Complex	(0.0, 0.0)
Logical	.FALSE.
Character (*len*)	*len* blanks

Result Type, Type Parameters, and Shape. The result is of the type and type parameters of VECTOR and it has rank two and shape [*n*, *n*] where *n* is the size of VECTOR.

Result Value. Element (*i*, *i*) of the result has value VECTOR (*i*) for $1 \le i \le n$. All other elements have the value FILL.

Example. DIAGONAL ([1, 2, 3]) has the value $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$.

13.9.29 DIGITS (X).

Description. Returns the number of significant digits in the model representing numbers of the same type and type parameters as the argument.

Kind. Inquiry function.

Argument. X must be of type integer or real. It may be scalar or array valued.

Result Type and Shape. Integer scalar.

Result Value. The result has value *q* if X is of type integer and *p* if X is of type real, where *q* and *p* are as defined in 13.5.1 for the model representing numbers of the same type and type parameters as X.

Example. DIGITS (X) has the value 24 for real X whose model is as at the end of 13.5.1.

13.9.30 DIM (X, Y).

Description. The difference X−Y if it is positive; otherwise zero.

Kind. Elemental function.

Arguments.

X	must be of type integer, real, or double precision.
Y	must be of the same type as X.

Result Type and Type Parameters. Same as X.

Result Value. The value of the result is X−Y if X > Y and zero otherwise.

Example. DIM (−3.0, 2.0) has the value 0.0.

13.9.31 DLBOUND (ARRAY, DIM).

Optional Argument. DIM

Description. Returns all the declared lower bounds of an array or a specified declared lower bound.

Kind. Inquiry function.

Arguments.

ARRAY	may be of any type. It must not be scalar. It must not be an allocatable array that is not allocated or an alias array that does not exist.
DIM (optional)	must be scalar and of type integer with value in the range $1 \le \text{DIM} \le n$, where *n* is the rank of ARRAY.

Result Type and Shape. The result is of type integer. It is scalar if DIM is present; otherwise, the result is an array of rank one and size *n*, where *n* is the rank of ARRAY.

Result Value.

Case (i): DLBOUND (ARRAY, DIM) has value equal to the declared lower bound for subscript DIM of ARRAY if dimension DIM of ARRAY does not have size zero and has the value 1 if dimension DIM has size zero. For an array section or an array expression, it has the value 1.

Case (ii): DLBOUND (ARRAY) has value whose *i*-th component is equal to DLBOUND (ARRAY, *i*), for *i* = 1, 2, ..., *n*, where *n* is the rank of ARRAY.

Example. If A is declared by the statement

```
REAL A (2:3, 7:10)
```

then DLBOUND (A) is [2, 7] and DLBOUND (A, DIM=2) is 7.

13.9.32 DUBOUND (ARRAY, DIM).

Optional Argument. DIM

Description. Returns all the declared upper bounds of an array or a specified declared upper bound.

Kind. Inquiry function.

Arguments.

ARRAY	may be of any type. It must not be scalar. It may not be an allocatable array that has not been allocated or an alias array that does not exist. If DIM is omitted or is present with value equal to the rank of ARRAY, ARRAY must not be an assumed-size array.
DIM (optional)	must be scalar and of type integer with value in the range $1 \leq$ DIM $\leq n$, where n is the rank of ARRAY.

Result Type and Shape. The result is of type integer. It is scalar if DIM is present; otherwise, the result is an array of rank one and size n, where n is the rank of ARRAY.

Result Value.

Case (i): DUBOUND (ARRAY, DIM) has value equal to the declared upper bound for subscript DIM of ARRAY if dimension DIM of ARRAY does not have size zero and has the value zero if dimension DIM has size zero. For an array section or an array expression, its value is the number of elements in the corresponding dimension.

Case (ii): DUBOUND (ARRAY) has value whose i-th component is equal to DUBOUND (ARRAY, i), for $i = 1, 2, ..., n$, where n is the rank of ARRAY.

Example. If A is declared by the statement

```
REAL A (2:3, 7:10)
```

then DUBOUND (A) is [3, 10] and DUBOUND (A, DIM=2) is 10.

13.9.33 DOTPRODUCT (VECTOR_A, VECTOR_B).

Description. Performs dot-product multiplication of numeric or Boolean vectors.

Kind. Transformational function.

Arguments.

VECTOR_A	must be of numeric type (integer, real, double precision, or complex) or of logical type. It must be array valued and of rank one.
VECTOR_B	must be of numeric type if VECTOR_A is of numeric type or of type logical if VECTOR_A is of type logical. It must be array valued and of rank one. It must be of the same size as VECTOR_A.

Result Type, Type Parameters, and Shape. If the arguments are of numeric type, the type and type parameters of the result are those of the expression VECTOR_A * VECTOR_B determined by the types of the arguments according to 7.1.4. If the arguments are those of the expression VECTOR_A * VECTOR_B as of type logical, the result is of type logical. The result is scalar.

Result Value.

Case (i): If VECTOR_A is of type integer, real, or double precision, the result has value SUM (VECTOR_A*VECTOR_B). If the vectors have size zero, the result has value zero.

Case (ii): If VECTOR_A is of type complex, the result has value SUM (CONJG (VECTOR_A)*VECTOR_B). If the vectors have size zero, the result has value zero.

Case (iii): If VECTOR_A is of type logical, the result has value ANY (VECTOR_A) .AND. VECTOR_B). If the vectors have size zero, the result has value .FALSE.

Example. DOTPRODUCT ([1, 2, 3], [2, 3, 4]) has the value 20.

13.9.34 DPROD (X, Y).

Description. Double precision product.

Kind. Elemental function.

Arguments.

X	must be of type real.
Y	must be of type real.

Result Type. Double precision.

Result Value. The value of the result is X * Y.

Example. DPROD (−3.0, 2.0) has the value −6.0D0.

13.9.35 DSHAPE (SOURCE).

Description. Returns the declared shape of an array or a scalar.

Kind. Inquiry function.

Argument. SOURCE may be of any type. It may be array valued or scalar.

Result Type and Shape. The result is an integer array of rank one whose size is equal to the rank of SOURCE.

Result Value. The value of the result is the declared shape of SOURCE.

Examples. The value of DSHAPE (A (2:5, −1:1)) is [4, 3]. The value of DSHAPE (3) is the null rank 1 array.

13.9.36 DSIZE (ARRAY, DIM).

Optional Argument. DIM

Description. Returns the declared extent of an array along a specified dimension or the total declared number of elements in the array.

Kind. Inquiry function.

Arguments.

ARRAY	may be of any type. It must not be scalar. If ARRAY is an assumed-size array, DIM must be present with value less than the rank of ARRAY.
DIM (optional)	must be scalar and of type integer with value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY.

Result Type and Shape. Integer scalar.

Result Value. The result has value equal to the declared extent of dimension DIM of ARRAY or, if DIM is absent, the total declared number of elements of ARRAY.

Examples. The value of DSIZE (A (2:5, −1:1), DIM=2) is 3. The value of DSIZE (A (2:5, −1:1)) is 12.

13.9.37 EFFECTIVE_EXPONENT_RANGE (X).

Description. Returns the decimal exponent range in the model representing numbers of the same type and type parameters as the argument.

Kind. Inquiry function.

Argument. X must be of type real or complex. It may be scalar or array valued.

Result Type and Shape. Integer scalar.

Result Value. The result has value INT (MIN (LOG10 (*huge*), −LOG10 (*tiny*))), where *huge* and *tiny* are the largest and smallest numbers in the model representing numbers of the same type and type parameters as X (see 13.5.1); *huge* has value HUGE (X) and *tiny* has value TINY (X).

Example. EFFECTIVE_EXPONENT_RANGE (X) has the value 38 for real X whose model is as at the end of 13.5.1, since in in this case *huge* = $(1-2^{-24}) \times 2^{127}$ and *tiny* = 2^{-127}.

13.9.38 EFFECTIVE_PRECISION (X).

Description. Returns the decimal precision in the model representing numbers of the same type and type parameters as the argument.

Kind. Inquiry function.

Argument. X must be of type real or complex. It may be scalar or array valued.

Result Type and Shape. Integer scalar.

Result Value. The result has value p if b is 10 and has value INT (($p-1$) * LOG10 (b)) otherwise, where b and p are as defined in 13.5.1 for the model representing numbers of the same type and type parameters as X.

Example. EFFECTIVE_PRECISION (X) has the value INT (23 * LOG10 (2.)) = INT (6.92...) = 6 for real X whose model is as at the end of 13.5.1.

13.9.39 ELBOUND (ARRAY, DIM).

Optional Argument. DIM

Description. Returns all the effective lower bounds of an array or a specified effective lower bound.

Kind. Inquiry function.

Arguments.

ARRAY	may be of any type. It must not be scalar. It must not be an allocatable array that is not allocated or an alias array that does not exist.
DIM (optional)	must be scalar and of type integer with value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY.

Result Type and Shape. The result is of type integer. It is scalar if DIM is present; otherwise, the result is an array of rank one and size n, where n is the rank of ARRAY.

Result Value.

Case (i): ELBOUND (ARRAY, DIM) has value equal to the effective lower bound for subscript DIM of ARRAY if dimension DIM of ARRAY does not have size zero and has the value 1 if dimension DIM has size zero. For an array section or an array expression, it has the value 1.

Case (ii): ELBOUND (ARRAY) has value whose *i*-th component is equal to ELBOUND (ARRAY, *i*), for *i* = 1, 2, ..., *n*, where *n* is the rank of ARRAY.

Example. If A is declared and its range is set as follows:

```
REAL, RANGE :: A (2:10, 5:10)
SET RANGE (4:6, 7:9) A
```

then ELBOUND (A) is [4, 7] and ELBOUND (A, DIM=2) is 7.

13.9.40 EUBOUND (ARRAY, DIM).

Optional Argument. DIM

Description. Returns all the effective upper bounds of an array or a specified effective upper bound.

Kind. Inquiry function.

Arguments.

ARRAY	may be of any type. It must not be scalar. It may not be an allocatable array that has not been allocated or an alias array that does not exist. If DIM is omitted or is present with value equal to the rank of ARRAY, ARRAY must not be an assumed-size array.
DIM (optional)	must be scalar and of type integer with value in the range 1 ≤ DIM ≤ *n*, where *n* is the rank of ARRAY.

Result Type and Shape. The result is of type integer. It is scalar if DIM is present; otherwise, the result is an array of rank one and size *n*, where *n* is the rank of ARRAY.

Result Value.

Case (i): EUBOUND (ARRAY, DIM) has value equal to the effective upper bound for subscript DIM of ARRAY if dimension DIM of ARRAY does not have size zero and has the value zero if dimension DIM has size zero. For an array section or an array expression, its value is the number of elements in the corresponding dimension.

Case (ii): EUBOUND (ARRAY) has value whose *i*-th component is equal to EUBOUND (ARRAY, *i*), for *i* = 1, 2, ..., *n*, where *n* is the rank of ARRAY.

Example. If A is declared by the statement

```
REAL A (2:3, 7:10)
```

then EUBOUND (A) is [3, 10] and EUBOUND (A, DIM=2) is 10.

13.9.41 ENABLED (CONDITION, LEVEL).

Optional Argument. LEVEL

Description. Determine whether a condition is enabled.

Kind. Inquiry function.

Arguments.

CONDITION	must be a condition name.
LEVEL (optional)	must be scalar and of type integer. Its value must not be negative. If omitted, the result is determined as though LEVEL were present with value 1.

Result Type and Shape. Logical scalar.

Result Value. The result is defined recursively, as follows:

Case (i): If the condition specified by CONDITION is enabled, the result is .TRUE.

Case (ii): If case (i) does not apply and either LEVEL is zero or the current program unit is a main program, the result is .FALSE.

Case (iii): If neither of the first two cases hold, the result is that of ENABLED (CONDITION, LEVEL$-$1) evaluated at the point of reference to the current program unit.

13.9.42 EOSHIFT (ARRAY, DIM, SHIFT, BOUNDARY).

Optional Argument. BOUNDARY

Description. Perform an end-off shift on an array expression of rank one or perform end-off shifts on all the complete rank-one sections along a given dimension of a many-ranked array expression. Elements are shifted off at one end of a section and copies of a boundary value are shifted in at the other end. Different sections may have different boundary values and may be shifted by different amounts and in different directions.

Kind. Transformational function.

Arguments.

ARRAY	may be of any type. It must not be scalar.
DIM	must be scalar and of type integer with value in the range $1 \le \text{DIM} \le n$, where n is the rank of ARRAY.
SHIFT	must be of type integer and must be scalar if ARRAY has rank one; otherwise, it must be scalar or of rank $n-1$ and of shape [E (1:DIM$-$1), E (DIM+1:n)], where E (1:n) is the shape of ARRAY.
BOUNDARY (optional)	must be of the same type and type parameters as ARRAY and must be scalar if ARRAY has rank one; otherwise, it must be either scalar or of rank $n-1$ and of shape [E (1:DIM-1), E (DIM+1:n)]. BOUNDARY may be omitted for the data types in the following table and, in this case, it is as if it were present with the scalar value shown.

Type of ARRAY	Value of BOUNDARY
Integer	0
Real	0.0
Double precision	0.0D0
Complex	(0.0, 0.0)
Logical	.FALSE.
Character (*len*)	*len* blanks

Result Type, Type Parameters, and Shape. The result has the type, type parameters, and shape of ARRAY.

Result Value. Element (s_1, s_2, ..., s_n) of the result has value that of ARRAY (s_1, s_2, ..., $s_{\mathrm{DIM}-1}$, $s_{\mathrm{DIM}}+sh$, $s_{\mathrm{DIM}+1}$, ..., s_n) where sh is SHIFT or SHIFT (s_1, s_2, ..., $s_{\mathrm{DIM}-1}$, $s_{\mathrm{DIM}+1}$, ..., s_n) provided the inequality $1 \le s_{\mathrm{DIM}} + sh = \mathrm{E}$ (DIM) holds and is otherwise BOUNDARY or BOUNDARY (s_1, s_2, ..., $s_{\mathrm{DIM}-1}$, $s_{\mathrm{DIM}+1}$, ..., s_n).

Examples.

Case (i): If V is the array [1, 2, 3, 4, 5, 6], the effect of shifting V end-off to the left by 3 positions is achieved by EOSHIFT (V, DIM = 1, SHIFT = 3) which has the value [4, 5, 6, 0, 0, 0]; EOSHIFT (V, DIM = 1, SHIFT = −2, BOUNDARY = 99) achieves an end-off shift to the right by 2 positions with the boundary value of 99 and has the value [99, 99, 1, 2, 3, 4].

Case (ii): The rows of an array of rank two may all be shifted by the same amount or by different amounts and the boundary elements can be the same or different. If M is the array $\begin{bmatrix} A & B & C \\ A & B & C \\ A & B & C \end{bmatrix}$, then the value of EOSHIFT (M, DIM = 2, SHIFT = −1, BOUNDARY = '*') is $\begin{bmatrix} * & A & B \\ * & A & B \\ * & A & B \end{bmatrix}$, and the value of CSHIFT (M, DIM = 2, SHIFT = [−1, 1, 0], BOUNDARY = ['*', '/', '?']) is $\begin{bmatrix} * & A & B \\ B & C & / \\ A & B & C \end{bmatrix}$

13.9.43 EPSILON (X).

Description. Returns a positive model number that is almost negligible compared to one in the model representing numbers of the same type and type parameters as the argument.

Kind. Inquiry function.

Argument. X must be of type real. It may be scalar or array valued.

Result Type, Type Parameters, and Shape. Scalar of the same type and type parameters as X.

Result Value. The result has value b^{1-p} where b and p are as defined in 13.5.1 for the model representing numbers of the same type and type parameters as X.

Example. EPSILON (X) has the value 2^{-23} for real X whose model is as at the end of 13.5.1.

13.9.44 ESHAPE (SOURCE).

Description. Returns the effective shape of an array or a scalar.

Kind. Inquiry function.

Argument. SOURCE may be of any type. It may be array valued or scalar.

Result Type and Shape. The result is an integer array of rank one whose size is equal to the rank of SOURCE.

Result Value. The value of the result is the effective shape of SOURCE.

Example. The value of ESHAPE (A (2:5, −1:1)) is [4, 3]. The value of ESHAPE (3) is the null rank 1 array.

13.9.45 ESIZE (ARRAY, DIM).

Optional Argument. DIM

Description. Returns the effective extent of an array along a specified dimension or the total effective number of elements in the array.

Kind. Inquiry function.

Arguments.

ARRAY	may be of any type. It must not be scalar. If ARRAY is an assumed-size array, DIM must be present with value less than the rank of ARRAY.
DIM (optional)	must be scalar and of type integer with value in the range $1 \le \text{DIM} \le n$, where n is the rank of ARRAY.

Result Type and Shape. Integer scalar.

Result Value. The result has value equal to the effective extent of dimension DIM of ARRAY or, if DIM is absent, the total effective number of elements of ARRAY.

Example. The value of ESIZE (A (2:5, −1:1), DIM=2) is 3. The value of ESIZE (A (2:5, −1:1)) is 12.

13.9.46 EXP (X).

Description. Exponential.

Kind. Elemental function.

Argument. X must be of type real, double precision, or complex.

Result Type and Type Parameters. Same as X.

Result Value. The result has value equal to a processor-dependent approximation to e^X. If X is of type complex, its imaginary part is regarded as a value in radians.

Example. EXP (1.0) has the value 2.7182818 (approximately).

13.9.47 EXPONENT (X).

Description. Returns the exponent part of the argument when represented as a model number.

Kind. Elemental function.

Argument. X must be of type real.

Result Type. Integer.

Result Value. The result has value equal to the exponent e of the model representation (see 13.5.1) for the value of X, provided X is nonzero and e is within range for integers.

Example. EXPONENT (1.0) has the value 1 for reals whose model is as at the end of 13.5.1.

13.9.48 FIRSTLOC (MASK, DIM).

Optional Argument. DIM

Description. Locate the leading edges of the set of true elements of a logical or bit mask.

Kind. Transformational function.

Arguments.

MASK	must be of type logical or bit. It must not be scalar. Its shape must be defined.
DIM (optional)	must be scalar and of type integer with value in the range $1 \le \text{DIM} \le n$, where n is the rank of MASK.

Result Type and Shape. The result is an array of the same shape as MASK and of type logical.

Result Value.

Case (i): The result of FIRSTLOC (MASK) has at most one true element. If MASK is all false, the result is all false. If MASK contains one or more true elements, the result has a single true element and it is in the position corresponding to the first true element (in subscript order value) in MASK.

Case (ii): The result of FIRSTLOC (MASK, DIM) is defined by applying FIRSTLOC to each of the one-dimensional array sections of MASK that lie parallel to dimension DIM. Thus, section $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$ of the result has value equal to FIRSTLOC (MASK $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$).

Examples. If MASK is $\begin{bmatrix} . & . & T & . \\ . & T & T & . \\ . & T & . & T \\ . & . & . & . \end{bmatrix}$, where "T" represents .TRUE. and "." represents .FALSE., then

Case (i): FIRSTLOC (MASK) is $\begin{bmatrix} . & . & . & . \\ . & T & . & . \\ . & . & . & . \\ . & . & . & . \end{bmatrix}$ and

Case (ii): FIRSTLOC (MASK, DIM = 1) is the "top-edge" $\begin{bmatrix} . & . & T & . \\ . & T & . & . \\ . & . & . & T \\ . & . & . & . \end{bmatrix}$.

13.9.49 FRACTION (X).

Description. Returns the fractional part of the model representation of the argument value.

Kind. Elemental function.

Argument. X must be of type real.

Result Type and Type Parameters. Same as X.

Result Value. The result has value $X \times b^{-e}$, where b and e are as defined in 13.5.1 for the model representation of X. If X has value zero, the result has value zero.

Example. FRACTION (3.0) has the value 0.75 for reals whose model is as at the end of 13.5.1.

13.9.50 HANDLED (CONDITION, LEVEL).

Optional Argument. LEVEL

Description. Determine whether a condition would be handled.

Kind. Inquiry function.

Arguments.

CONDITION	must be a condition name.
LEVEL (optional)	must be scalar and of type integer. Its value must not be negative. If omitted, the result is determined as though LEVEL were present with value HUGE (0).

Result Type and Shape. Logical scalar.

Result Value. The result is defined recursively as follow:

Case (i): If a handler is supplied for an occurrence of the condition specified by CONDITION, the result is .TRUE.

Case (ii): If no such handler is supplied and either LEVEL is zero or the current program unit is a main program, the result is .FALSE.

Case (iii): If neither of the first two cases hold, the result is that of HANDLED (CONDITION, LEVEL-1) evaluated at the point of reference to the current program unit.

13.9.51 HUGE (X).

Description. Returns the largest number in the model representing numbers of the same type and type parameters as the argument.

Kind. Inquiry function.

Argument. X must be of type integer or real. It may be scalar or array valued.

Result Type, Type Parameters, and Shape. Scalar of the same type and type parameters as X.

Result Value. The result has value $r^q - 1$ if X is of type integer and $(1 - b^{-p})b^{e_{max}}$ if X is of type real, where r, q, b, p, and e_{max} are as defined in 13.5.1 for the model representing numbers of the same type and type parameters as X.

Example. HUGE (X) has the value $(1 - 2^{-24}) \times 2^{127}$ for real X whose model is as at the end of 13.5.1.

13.9.52 IACHAR (C).

Description. Returns the position of a character in the ASCII collating sequence.

Kind. Elemental function.

Argument. C must be of type character and of length one.

Result Type. Integer.

Result Value. The result is the position of C in the collating sequence described in ANSI X3.4-1977 (ASCII). It satisfies the inequality ($0 \le$ IACHAR (C) ≤ 127). A processor-dependent value is returned if C is not in the ASCII collating sequence. The results must be consistent with the LGE, LGT, LLE, and LLT lexical comparison functions. For example, if LLE (C, D) is true, IACHAR (C) .LE. IACHAR (D) is true where C and D are any two characters representable by the processor.

Example. IACHAR ('X') has the value 88.

13.9.53 IBITLR (B).

Description. Convert a bit array to an integer, counting left to right.

Kind. Transformational function.

Argument. B must be of type bit and rank one. Its size must satisfy the inequality SIZE (B) $\leq$ MAXBITS (1).

Result Type and Shape. Scalar integer.

Result Value. The result has value equal to the integer represented by the bits in the array B, regarded as a bit string with the element having the largest subscript value being the least significant bit of the result. IBITLR (BITLR (J)) must have the value J for any value of the integer J. BITLR (IBITLR (B), SIZE (B)) must have the value B for any value of a bit array B for which SIZE (B) $\leq$ MAXBITS (1).

Example. IBITLR ([B'0', B'1', B'0', B'1']) has the value 5.

13.9.54 IBITRL (B).

Description. Convert a bit array to an integer, counting right to left.

Kind. Transformational function.

Argument. B must be of type bit and rank one. Its size must satisfy the inequality SIZE (B) $\leq$ MAXBITS (1).

Result Type and Shape. Scalar integer.

Result Value. The result has value equal to the integer represented by the bits in the array B, regarded as a bit string with the element having the largest subscript value being the most significant bit of the result. IBITRL (BITRL (J)) must have the value J for any value of the integer J. BITRL (IBITRL (B), SIZE (B)) must have the value B for any value of a bit array B for which SIZE (B) $\leq$ MAXBITS (1).

Example. IBITRL ([B'1', B'0', B'1', B'0']) has the value 5.

13.9.55 ICHAR (C).

Description. Returns the position of a character in the processor collating sequence.

Kind. Elemental function.

Argument. C must be of type character and of length one. Its value must be that of a character capable of representation in the processor.

Result Type. Integer.

Result Value. The result is the position of C in the processor collating sequence and is in the range $0 \leq$ ICHAR (C) $\leq n-1$, where n is the number of characters in the collating sequence. For any characters C and D capable of representation in the processor, C .LE. D is true if and only if ICHAR (C) .LE. ICHAR (D) is true and C .EQ. D is true if and only if ICHAR (C). EQ. ICHAR (D) is true.

Example. ICHAR ('X') has the value 88 on a processor using the ASCII collating sequence.

13.9.56 INDEX (STRING, SUBSTRING).

Description. Returns the starting position of a substring within a string.

Kind. Elemental function.

Arguments.

STRING	must be of type character.
SUBSTRING	must be of type character.

Result Type. Integer.

Result Value. If SUBSTRING occurs within STRING, the value returned is the minimum value of I such that STRING (I : I + LEN (SUBSTRING) − 1) = = SUBSTRING; otherwise, zero is returned. Zero is returned if LEN (STRING) < LEN (SUBSTRING) and one is returned if LEN (SUBSTRING) = 0.

Example. INDEX ('FORTRAN', 'R') has value 3.

13.9.57 INT (A).

Description. Convert to integer type.

Kind. Elemental function.

Argument. A must be of type integer, real, double precision, or complex.

Result Type. Integer.

Result Value.

Case (i): If A is of type integer, INT (A) = A.

Case (ii): If A is of type real or double precision, there are two cases: if $|A| < 1$, INT (A) has the value 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.

Case (iii): If A is of type complex, INT (A) is the value obtained by applying the case (ii) rule to the real part of A.

Example. INT(−3.7) has the value −3.

13.9.58 ISCAN (STRING, SET).

Description. Scan a string for a character in a set of characters.

Kind. Elemental function.

Arguments.

STRING	must be of type character.
SET	must be of type character.

Result Type. Integer.

Result Value. If any of the characters of SET appears in STRING, the value of the result is the integer index of the leftmost character of STRING that is in SET. The result is zero if STRING does not contain any of the characters that are in SET or if the length of STRING or SET is zero.

Example. ISCAN ('FORTRAN', 'TR') has value 3.

13.9.59 LASTLOC (MASK, DIM).

Optional Argument. DIM

Description. Locate the trailing edges of the set of true elements of a logical or bit mask.

Kind. Transformational function.

Arguments.

MASK	must be of type logical or bit. It must not be scalar.
DIM (optional)	must be scalar and of type integer with value in the range $1 \le \text{DIM} \le n$, where n is the rank of MASK.

Result Type and Shape. The result is an array of the same shape as MASK and of type logical???.

Result Value.

Case (i): The result of LASTLOC (MASK) has at most one true element. If MASK is all false, the result is all false. If MASK contains one or more true elements, the result has a single true element and it is in the position corresponding to the last true element (in subscript order value) in MASK.

Case (ii): The result of LASTLOC (MASK, DIM) is defined by applying LASTLOC to each of the one-dimensional array sections of MASK that lie parallel to dimension DIM. Thus, section (s_1, s_2, ..., $s_{\text{DIM}-1}$, :, $s_{\text{DIM}+1}$, ..., s_n) of the result has value equal to LASTLOC (MASK (s_1, s_2, ..., $s_{\text{DIM}-1}$, :, $s_{\text{DIM}+1}$, ..., s_n)).

Examples. If MASK is $\begin{bmatrix} . & . & T & . \\ . & T & T & . \\ . & T & . & T \\ . & . & . & . \end{bmatrix}$, where "T" represents .TRUE. and "." represents .FALSE., then

Case (i): LASTLOC (MASK) is $\begin{bmatrix} . & . & . & . \\ . & . & . & . \\ . & . & . & T \\ . & . & . & . \end{bmatrix}$ and

Case (ii): LASTLOC (MASK, DIM=2) is $\begin{bmatrix} . & . & T & . \\ . & . & T & . \\ . & . & . & T \\ . & . & . & . \end{bmatrix}$.

13.9.60 LBIT (B).

Description. Convert bit to logical type.

Kind. Elemental function.

Argument. B must be of type bit.

Result Type. Logical.

Result Value. The result has the value .TRUE. if B has the value B'1' and the value .FALSE. if B has the value B'0'.

Example. LBIT (B'1') has the value .TRUE.

13.9.61 LEN (STRING).

Description. Returns the length of a character entity.

Kind. Inquiry function.

Argument. STRING must be of type character. It may be scalar or array valued.

Result Type and Shape. Integer scalar.

Result Value. The result has value equal to the number of characters in STRING if it is scalar or in a component of STRING if it is array valued.

Example. If C is declared by the statement

```
CHARACTER (11) C (100)
```

LEN (C) has value 11.

13.9.62 LEN_TRIM (STRING).

Description. Returns the length of the character argument without trailing blank characters.

Kind. Elemental function.

Argument. STRING must be of type character.

Result Type. Integer.

Result Value. The result has a value equal to the number of characters before any trailing blanks in STRING are removed. If the argument contains no nonblank characters, the result is zero.

Example. LEN_TRIM (' A B ') has value 4 and LEN_TRIM (' ') has value 0.

13.9.63 LGE (STRING_A, STRING_B).

Description. Test whether a string is lexically greater than or equal to another string, based on the ASCII collating sequence.

Kind. Elemental function.

Arguments.

STRING_A	must be of type character.
STRING_B	must be of type character.

Result Type. Logical.

Result Value. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string. If either string contains a character not in the ASCII character set, the result is processor dependent. The result is true if the strings are equal or if STRING_A follows STRING_B in the collating sequence described in ANSI X3.4-1977 (ASCII); otherwise, the result is false.

Example. LGE ('ONE', 'TWO') has the value .FALSE.

13.9.64 LGT (STRING_A, STRING_B).

Description. Test whether a string is lexically greater than another string, based on the ASCII collating sequence.

Kind. Elemental function.

Arguments.

STRING_A	must be of type character.
STRING_B	must be of type character.

Result Type. Logical.

Result Value. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string. If either string contains a character not in the ASCII character set, the result is processor dependent. The result is true if STRING_A follows STRING_B in the collating sequence described in ANSI X3.4-1977 (ASCII); otherwise, the result is false.

Example. LGT ('ONE', 'TWO') has the value .FALSE.

13.9.65 LLE (STRING_A, STRING_B).

Description. Test whether a string is lexically less than or equal to another string, based on the ASCII collating sequence.

Kind. Elemental function.

Arguments.

STRING_A	must be of type character.
STRING_B	must be of type character.

Result Type. Logical.

Result Value. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string. If either string contains a character not in the ASCII character set, the result is processor dependent. The result is true if the strings are equal or if STRING_A precedes STRING_B in the collating sequence described in ANSI X3.4-1977 (ASCII); otherwise, the result is false.

Example. LLE ('ONE', 'TWO') has the value .TRUE.

13.9.66 LLT (STRING_A, STRING_B).

Description. Test whether a string is lexically less than another string, based on the ASCII collating sequence.

Kind. Elemental function.

Arguments.

STRING_A	must be of type character.
STRING_B	must be of type character.

Result Type. Logical.

Result Value. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string. If either string contains a character not in the ASCII character set, the result is processor dependent. The result is true if STRING_A precedes STRING_B in the collating sequence described in ANSI X3.4-1977 (ASCII); otherwise, the result is false.

Example. LLT ('ONE', 'TWO') has the value .TRUE.

13.9.67 LOG (X).

Description. Natural logarithm.

Kind. Elemental function.

Argument. X must be of type real, double precision, or complex. Unless X is complex, its value must be greater than zero. If X is complex, its value must not be zero.

Result Type and Type Parameters. Same as X.

Result Value. The result has value equal to a processor-dependent approximation to $\log_e X$. A result of type complex is the principal value with imaginary part ω in the range $-\pi < \omega \leq \pi$. The imaginary part of the result is π only when the real part of the argument is less than zero and the imaginary part of the argument is zero.

Example. LOG (10.0) has the value 2.3025851 (approximately).

13.9.68 LOG10 (X).

Description. Common logarithm.

Kind. Elemental function.

Argument. X must be of type real or double precision. The value of X must be greater than zero.

Result Type and Type Parameters. Same as X.

Result Value. The result has value equal to a processor-dependent approximation to $\log_{10}$X.

Example. LOG10 (10.0) has the value 1.0 (approximately).

13.9.69 MATMUL (MATRIX_A, MATRIX_B).

Description. Performs matrix multiplication of numeric or Boolean matrices.

Kind. Transformational function.

Arguments.

MATRIX_A	must be of numeric type (integer, real, double precision, or complex) or of logical type. It must be array valued and of rank one or two. Its shape must be defined.
MATRIX_B	must be of numeric type if MATRIX_A is of numeric type and of logical type if MATRIX_A is of logical type. It must be array valued and of rank one or two. If MATRIX_A has rank one, MATRIX_B must have rank two. Its shape must be defined. The size of the first (or only) dimension of MATRIX_B must equal the size of the last (or only) dimension of MATRIX_A.

Result Type, Type Parameters, and Shape. If the arguments are of numeric type, the type and type parameters of the result are determined by the types of the arguments according to 7.1.4. If the arguments are of type logical, the result is of type logical. The shape of the result depends on the shapes of the arguments as follows:

Case (i): If MATRIX_A has shape [n, m] and MATRIX_B has shape [m, k], the result has shape [n, k].

Case (ii): If MATRIX_A has shape [m] and MATRIX_B has shape [m, k], the result has shape [k].

Case (iii): If MATRIX_A has shape [*n*, *m*] and MATRIX_B has shape [*m*], the result has shape [*n*].

Result Value.

Case (i): Element (*i*, *j*) of the result has value SUM (MATRIX_A (*i*, :) * MATRIX_B (:, *j*)) if the arguments are of numeric type and has value ANY (MATRIX_A (*i*, :) .AND. MATRIX_B (:, *j*)) if the arguments are of logical type.

Case (ii): Element (*j*) of the result has value SUM (MATRIX_A (:) * MATRIX_B (:, *j*)) if the arguments are of numeric type and has value ANY (MATRIX_A (:) .AND. MATRIX_B (:, *j*)) if the arguments are of logical type.

Case (iii): Element (*i*) of the result has value SUM (MATRIX_A (*i*, :) * MATRIX_B (:)) if the arguments are of numeric type and has value ANY (MATRIX_A (*i*, :) .AND. MATRIX_B (:)) if the arguments are of logical type.

Examples. Let A and B be the matrices $\begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \end{bmatrix}$ and $\begin{bmatrix} 1 & 2 \\ 2 & 3 \\ 3 & 4 \end{bmatrix}$; let X and Y be the vectors [1, 2] and [1, 2, 3].

Case (i): The result of MATMUL (A, B) is the matrix-matrix product AB with value $\begin{bmatrix} 14 & 20 \\ 20 & 29 \end{bmatrix}$.

Case (ii): The result of MATMUL (X, A) is the vector-matrix product XA with value [5, 8, 11].

Case (iii): The result of MATMUL (A, Y) is the matrix-vector product AY with value [14, 20].

13.9.70 MAX (A1, A2, A3, ...).

Optional Arguments. A3, ...

Description. Maximum value.

Kind. Elemental function.

Arguments. The arguments must all have the same type which must be integer, real, or double precision and they must all have the same type parameters.

Result Type and Type Parameters. Same as the arguments.

Result Value. The value of the result is that of the largest argument.

Example. MAX (−9.0, 7.0, 2.0) has the value 7.0.

13.9.71 MAXBITS (I).

Description. Returns the maximum size of a bit array that can be converted to a value of type integer.

Kind. Inquiry function.

Argument. I must be of type integer.

Result Type and Shape. Integer scalar.

Result Value. The result has value equal to the maximum size of a bit array B that can be converted to integer using IBITLR (B) or IBITRL (B).

13.9.72 MAXEXPONENT (X).

Description. Returns the maximum exponent in the model representing numbers of the same type and type parameters as the argument.

Kind. Inquiry function.

Argument. X must be of type real. It may be scalar or array valued.

Result Type and Shape. Integer scalar.

Result Value. The result has value e_{max}, as defined in 13.5.1 for the model representing numbers of the same type and type parameters as X.

Example. MAXEXPONENT (X) has the value 127 for real X whose model is as at the end of 13.5.1.

13.9.73 MAXLOC (ARRAY, MASK).

Optional Argument. MASK

Description. Determine the location of an element of ARRAY having the maximum value of the elements identified by MASK.

Kind. Transformational function.

Arguments.

ARRAY	must be of type integer, real, or double precision. It must not be scalar.
MASK (optional)	must be of type logical or bit and must be conformable with ARRAY.

Result Type and Shape. The result is of type integer; it is an array of rank one and of size equal to the rank of ARRAY.

Result Value.

Case (i): If MASK is absent, the result is a rank-one array whose element values are the values of the subscripts (in subscript order value) of an element of ARRAY whose value equals the maximum value of all of the elements of ARRAY. The *i*th subscript returned lies in the range 1 to e_i, where e_i is the extent of the *i*th dimension of ARRAY. If more than one element has maximum value, the element whose subscripts are returned is processor dependent. If ARRAY has size zero, the value of the result is processor dependent.

Case (ii): If MASK is present, the result is a rank-one array whose element values are the values of the subscripts (in subscript order value) of an element of ARRAY, corresponding to a true element of MASK, whose value equals the maximum value of all such elements of ARRAY. The *i*th subscript returned lies in the range 1 to e_i, where e_i is the extent of the *i*th dimension of ARRAY. If more than one such element has maximum value, the element whose subscripts are returned is processor dependent. If there are no such elements (that is, if ARRAY has size zero or every component of MASK has the value .FALSE.), the value of the result is processor dependent.

Examples.

Case (i): The value of MAXLOC ([2, 4, 6]) is [3].

Case (ii): If A has the value $\begin{bmatrix} 0 & -5 & 8 & -3 \\ 3 & 4 & -1 & 2 \\ 1 & 5 & 6 & -4 \end{bmatrix}$, MAXLOC (A, MASK=A.LT.6) has the value [3, 2].

13.9.74 MAXVAL (ARRAY, DIM, MASK).

Optional Arguments. DIM, MASK

Description. Maximum value of the elements of ARRAY along dimension DIM corresponding to the true elements of MASK.

Kind. Transformational function.

Arguments.

ARRAY	must be of type integer, real, or double precision. It must not be scalar. Its shape must be defined.
DIM (optional)	must be scalar and of type integer with value in the range $1 \le \text{DIM} \le n$, where n is the rank of ARRAY.
MASK	(optional) must be of type logical or bit and must be conformable with ARRAY.

Result Type, Type Parameters, and Shape. The result is of the same type and type parameters as ARRAY. It is scalar if DIM is absent or ARRAY has rank one; otherwise, the result is an array of rank $n-1$ and of shape $(d_1, d_2, ..., d_{\text{DIM}-1}, d_{\text{DIM}+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of ARRAY.

Result Value.

Case (i): The result of MAXVAL (ARRAY) has value equal to the maximum value of all the elements of ARRAY or has value −HUGE (ARRAY) if ARRAY has size zero.

Case (ii): The result of MAXVAL (ARRAY, MASK) has value equal to the maximum value of the elements of ARRAY corresponding to true elements of MASK or has value −HUGE (ARRAY) if there are no true elements.

Case (iii): If ARRAY has rank one, MAXVAL (ARRAY, DIM [,MASK]) has value equal to that of MAXVAL (ARRAY [,MASK]). Otherwise, the value of element $(s_1, s_2, ..., s_{\text{DIM}-1}, s_{\text{DIM}+1}, ..., s_n)$ of MAXVAL (ARRAY, DIM [,MASK]) is equal to MAXVAL (ARRAY $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$, [, MASK $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$]).

Examples.

Case (i): The value of MAXVAL ([1, 2, 3]) is 3.

Case (ii): MAXVAL (C, MASK = C .GT. 0.0) finds the maximum of the positive elements of C.

Case (iii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$, MAXVAL (B, DIM=1) is [2, 4, 6] and MAXVAL (B, DIM=2) is [5, 6].

13.9.75 MERGE (TSOURCE, FSOURCE, MASK).

Description. Choose alternative value according to value of a mask.

Kind. Elemental function.

Arguments.

TSOURCE	may be of any type.
FSOURCE	must be of the same type and type parameters as TSOURCE.
MASK	must be of type logical or bit.

Result Type and Type Parameters. Same as TSOURCE.

Result Value. The result is TSOURCE if MASK is true and FSOURCE otherwise.

Example. If TSOURCE is the array $\begin{bmatrix} 1 & 6 & 5 \\ 2 & 4 & 6 \end{bmatrix}$, FSOURCE is the array $\begin{bmatrix} 0 & 3 & 2 \\ 7 & 4 & 8 \end{bmatrix}$ and MASK is the array $\begin{bmatrix} \mathrm{T} & . & \mathrm{T} \\ . & . & \mathrm{T} \end{bmatrix}$, where "T" represents .TRUE. and "." represents .FALSE., then MERGE (TSOURCE, FSOURCE, MASK) is $\begin{bmatrix} 1 & 3 & 5 \\ 7 & 4 & 6 \end{bmatrix}$.

13.9.76 MIN (A1, A2, A3, ...).

Optional Arguments. A3, ...

Description. Minimum value.

Kind. Elemental function.

Arguments. The arguments must all be of the same type which must be integer, real, or double precision and they must all have the same type parameters.

Result Type and Type Parameters. Same as the arguments.

Result Value. The value of the result is that of the smallest argument.

Example. MIN (−9.0, 7.0, 2.0) has the value −9.0.

13.9.77 MINEXPONENT (X).

Description. Returns the minimum (most negative) exponent in the model representing numbers of the same type and type parameters as the argument.

Kind. Inquiry function.

Argument. X must be of type real. It may be scalar or array valued.

Result Type and Shape. Integer scalar.

Result Value. The result has value $e_{\min}$, as defined in 13.5.1 for the model representing numbers of the same type and type parameters as X.

Example. MINEXPONENT (X) has the value −126 for real X whose model is as at the end of 13.5.1.

13.9.78 MINLOC (ARRAY, MASK).

Optional Argument. MASK

Description. Determine the location of an element of ARRAY having the minimum value of the elements identified by MASK.

Kind. Transformational function.

Arguments.

ARRAY	must be of type integer, real, or double precision. It must not be scalar.

MASK (optional)	must be of type logical or bit and must be conformable with ARRAY.

Result Type and Shape. The result is of type integer; it is an array of rank one and of size equal to the rank of ARRAY.

Result Value.

Case (i): If MASK is absent, the result is a rank-one array whose element values are the values of the subscripts (in subscript order value) of an element of ARRAY whose value equals the minimum value of all the elements of ARRAY. The *i*th subscript returned lies in the range 1 to e_i, where e_i is the extent of the *i*th dimension of ARRAY. If more than one element has minimum value, the element whose subscripts are returned is processor dependent. If ARRAY has size zero, the value of the result is processor dependent.

Case (ii): If MASK is present, the result is a rank-one array whose element values are the values of the subscripts (in subscript order value) of an element of ARRAY, corresponding to a true element of MASK, whose value equals the minimum value of all such elements of ARRAY. The *i*th subscript returned lies in the range 1 to e_i, where e_i is the extent of the *i*th dimension of ARRAY. If more than one such element has minimum value, the element whose subscripts are returned is processor dependent. If ARRAY has size zero or every element of MASK has the value .FALSE., the value of the result is processor dependent.

Examples.

Case (i): The value of MINLOC ([2, 4, 6]) is [1].

Case (ii): If A has the value $\begin{bmatrix} 0 & -5 & 8 & -3 \\ 3 & 4 & -1 & 2 \\ 1 & 5 & 6 & -4 \end{bmatrix}$, MINLOC (A, MASK = A.GT. −4) has the value [1,4].

13.9.79 MINVAL (ARRAY, DIM, MASK).

Optional Arguments. DIM, MASK

Description. Minimum value of all the elements of ARRAY along dimension DIM corresponding to true elements of MASK.

Kind. Transformational function.

Arguments.

ARRAY	must be of type integer, real, or double precision. It must not be scalar.
DIM (optional)	must be scalar and of type integer with value in the range $1 \le \text{DIM} \le n$, where n is the rank of ARRAY.
MASK	(optional) must be of type logical or bit and must be conformable with ARRAY.

Result Type, Type Parameters, and Shape. The result is of the same type and type parameters as ARRAY. It is scalar if DIM is absent or ARRAY has rank one; otherwise, the result is an array of rank $n-1$ and of shape $(d_1, d_2, ..., d_{\text{DIM}-1}, d_{\text{DIM}+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of ARRAY.

Result Value.

Case (i): The result of MINVAL (ARRAY) has value equal to the minimum value of all the elements of ARRAY or has value HUGE (ARRAY) if ARRAY has size zero.

Case (ii): The result of MINVAL (ARRAY, MASK) has value equal to the minimum value of the elements of ARRAY corresponding to true elements of MASK or has value HUGE (ARRAY) if there are no true elements.

Case (iii): If ARRAY has rank one, MINVAL (ARRAY, DIM [,MASK]) has value equal to that of MINVAL (ARRAY [,MASK]). Otherwise, the value of element (s_1, s_2, ..., $s_{\mathrm{DIM}-1}$, $s_{\mathrm{DIM}+1}$, ..., s_n) of MINVAL (ARRAY, DIM [,MASK]) is equal to MINVAL (ARRAY (s_1, s_2, ..., $s_{\mathrm{DIM}-1}$, :, $s_{\mathrm{DIM}+1}$, ..., s_n) [, MASK (s_1, s_2, ..., $s_{\mathrm{DIM}-1}$, :, $s_{\mathrm{DIM}+1}$, ..., s_n)]).

Examples.

Case (i): The value of MINVAL ([1, 2, 3]) is 1.

Case (ii): MINVAL (C, MASK = C .GT. 0.0) forms the minimum of the positive elements of C.

Case (iii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$, MINVAL (B, DIM=1) is [1, 3, 5] and MINVAL (B, DIM=2) is [1, 2].

13.9.80 MOD (A, P).

Description. Remainder modulo P.

Kind. Elemental function.

Arguments.

A	must be of type integer, real, or double precision.
P	must be of the same type as A.

Result Type and Type Parameters. Same as A.

Result Value. If P $\neq$ 0, the value of the result is A$-$INT (A/P) $*$ P. If P = 0, the result is undefined.

Example. MOD (3.0, 2.0) has the value 1.0.

13.9.81 NEAREST (X, S).

Description. Returns the nearest different machine representable number in a given direction.

Kind. Elemental function.

Arguments.

X	must be of type real.
S	must be of type real and not equal to zero.

Result Type and Type Parameters. Same as X.

Result Value. The result has value equal to the machine representable number distinct from X and nearest to it in the direction of the infinity with the same sign as S.

Example. NEAREST (3.0, 2.0) has the value $3+2^{-22}$ on a machine whose representation is that of the model at the end of 13.5.1.

13.9.82 NINT (A).

Description. Nearest integer.

Kind. Elemental function.

Argument. A must be of type real or double precision.

Result Type. Integer.

Result Value. If A > 0, NINT (A) has the value INT (A+0.5); if A ≤ 0, NINT (A) has the value INT (A−0.5).

Example. NINT (2.783) has the value 3.

13.9.83 PACK (ARRAY, MASK, VECTOR).

Optional Argument. VECTOR

Description. Pack an array into an array of rank one under the control of a mask.

Kind. Transformational function.

Arguments.

ARRAY	may be of any type. It must not be scalar.
MASK	must be of type logical or bit and must be conformable with ARRAY.
VECTOR (optional)	must be of the same type and type parameters as ARRAY and must have rank one. It must have at least as many elements as there are true elements in MASK and if MASK is scalar with value true, it must have at least as many elements as there are in ARRAY.

Result Type, Type Parameters, and Shape. The result is an array of rank one with the same type and type parameters as ARRAY. If VECTOR is present, the result size is that of VECTOR; otherwise, the result size is the number *t* of true elements in MASK unless MASK is scalar with value true, in which case the result size is the size of ARRAY.

Result Value. Element *i* of the result is the *i*-th element of ARRAY that corresponds to a true element of MASK, taking elements in subscript order value, for $i = 1, 2, ..., t$. If VECTOR is present and has size $n > t$, element *i* of the result has value VECTOR (*i*), for $i = t + 1, ..., n$.

Example. The nonzero elements of an array M with value $\begin{bmatrix} 0 & 0 & 0 \\ 9 & 0 & 0 \\ 0 & 0 & 7 \end{bmatrix}$ may be "gathered" by the function PACK. The result of PACK (M, MASK=M.NE.0) is [9, 7] and the result of PACK (M, M.NE.0, VECTOR=[6[0]]) is [9, 7, 0, 0, 0, 0].

13.9.84 PRESENT (A).

Description. Determine whether an optional argument is present.

Kind. Inquiry function

Argument. A must be an optional argument of the procedure in which the PRESENT function reference appears.

Result Type and Shape. Logical scalar.

Result Value. The result has the value .TRUE. if A is present and is otherwise .FALSE.

13.9.85 PRODUCT (ARRAY, DIM, MASK).

Optional Arguments. DIM, MASK

Description. Product of all the elements of ARRAY along dimension DIM corresponding to the true elements of MASK.

Kind. Transformational function.

Arguments.

ARRAY	must be of type integer, real, double precision, or complex. It must not be scalar. Its shape must be defined.
DIM (optional)	must be scalar and of type integer with value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY.
MASK	(optional) must be of type logical or bit and must be conformable with ARRAY.

Result Type, Type Parameters, and Shape. The result is of the same type and type parameters as ARRAY. It is scalar if DIM is absent or ARRAY has rank one; otherwise, the result is an array of rank $n-1$ and of shape $(d_1, d_2, ..., d_{\text{DIM}-1}, d_{\text{DIM}+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of ARRAY.

Result Value.

Case (i): The result of PRODUCT (ARRAY) has value equal to a processor-dependent approximation to the product of all the elements of ARRAY or has value one if ARRAY has size zero.

Case (ii): The result of PRODUCT (ARRAY, MASK) has value equal to a processor-dependent approximation to the product of the elements of ARRAY corresponding to true elements of MASK or has value one if there are no true elements.

Case (iii): If ARRAY has rank one, PRODUCT (ARRAY, DIM [,MASK]) has value equal to that of PRODUCT (ARRAY [,MASK]). Otherwise, the value of element $(s_1, s_2, ..., s_{\text{DIM}-1}, s_{\text{DIM}+1}, ..., s_n)$ of PRODUCT (ARRAY, DIM [,MASK]) is equal to PRODUCT (ARRAY $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$ [, MASK $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$]).

Examples.

Case (i): The value of PRODUCT ([1, 2, 3]) is 6.

Case (ii): PRODUCT (C, MASK = C .GT. 0.0) forms the product of the positive elements of C.

Case (iii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$, PRODUCT (B, DIM = 1) is [2, 12, 30] and PRODUCT (B, DIM = 2) is [15, 48].

13.9.86 PROJECT (ARRAY, MASK, FIELD, DIM).

Optional Argument. DIM

Description. Select masked values from an array.

Kind. Transformational function.

Arguments.

ARRAY	may be of any type. It must not be scalar. Its shape must be defined.
MASK	must be of type logical or bit and of the same shape as ARRAY. If DIM is absent, MASK must have at most one true element; otherwise, each section MASK (s_1, ..., $s_{\text{DIM}-1}$, :, $s_{\text{DIM}+1}$, ..., s_n) must have at most one true element.
FIELD	must be of the same type and type parameters as ARRAY. It must be scalar if DIM is absent. If DIM is present, FIELD must have rank $n-1$ and shape [E (1:DIM−1), E (DIM+1:n)], where E (1:n) is the shape of ARRAY.
DIM (optional)	must be scalar and of type integer with value in the range $1 \le \text{DIM} \le n$, where n is the rank of ARRAY.

Result Type, Type Parameters, and Shape. The result is of the type and type parameters of ARRAY. It is scalar if DIM is absent or ARRAY has rank one; otherwise, the result has rank $n-1$ and shape [E (1:DIM−1), E (DIM+1:n)] where E (1:n) is the shape of ARRAY.

Result Value.

Case (i): The result of PROJECT (ARRAY, MASK, FIELD) is the element of ARRAY corresponding to the true element of MASK if there is one and is FIELD otherwise. Note that if MASK has zero size, the result has value FIELD.

Case (ii): If ARRAY has rank one, PROJECT (ARRAY, MASK, FIELD, DIM) has value equal to that of PROJECT (ARRAY, MASK, FIELD). Otherwise, the value of element (s_1, ..., $s_{\text{DIM}-1}$, $s_{\text{DIM}+1}$, ..., s_n) of PROJECT (ARRAY, MASK, FIELD, DIM) is equal to PROJECT (ARRAY (s_1, ..., $s_{\text{DIM}-1}$, :, $s_{\text{DIM}+1}$, ..., s_n), MASK (s_1, ..., $s_{\text{DIM}-1}$, :, $s_{\text{DIM}+1}$, ..., s_n), FIELD (s_1, ..., $s_{\text{DIM}-1}$, $s_{\text{DIM}+1}$, ..., s_n)). Note that if ARRAY (and MASK) have size zero because E (DIM) has value zero, the result may have nonzero size with all its values coming from FIELD.

Examples.

Case (i): If V is the array [1, 2, 3, 4] and P is the mask [., ., T, .], where "T" represents .TRUE. and "." represents .FALSE., the value of PROJECT (V, MASK=P, FIELD=0) is the scalar 3, and the value of PROJECT (V, MASK=V.GT.5, FIELD=99) is the scalar 99. If A is the array $\begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix}$ and L is the array $\begin{bmatrix} . & . & . & . \\ . & . & T & . \\ . & . & . & . \end{bmatrix}$, the value of PROJECT (A, MASK=L, FIELD=0) is the scalar 8.

Case (ii): Using the arrays of case (i), the value of PROJECT (A, L, [0, 0, 0], DIM=2) is the array [0, 8, 0], and the value of PROJECT (A, L, [0, 0, 0, 0], DIM=1) is the array [0, 0, 8, 0].

The first nonzero number in each column of the table TABLE = $\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 3 & 5 & 0 \\ 1 & 4 & 6 & 0 \end{bmatrix}$ is located by the mask M = $\begin{bmatrix} . & . & . & . \\ . & T & . & . \\ . & . & T & . \\ T & . & . & . \end{bmatrix}$. A vector which contains those nonzero numbers can be extracted from TABLE by the PROJECT function. Thus, the value of PROJECT (TABLE, M, [−1, −1, −1, −1], DIM=1) is that vector, namely [1, 2, 5, −1]. Note that M itself is the value of FIRSTLOC (TABLE.NE.0, DIM = 1).

13.9.87 RADIX (X).

Description. Returns the base of the model representing numbers of the same type and type parameters as the argument.

Kind. Inquiry function.

Argument. X must be of type integer or real. It may be scalar or array valued.

Result Type and Shape. Integer scalar.

Result Value. The result has value *r* if X is of type integer and *b* if X is of type real, where *r* and *b* are as defined in 13.5.1 for the model representing numbers of the same type and type parameters as X.

Example. RADIX (X) has the value 2 for real X whose model is as at the end of 13.5.1.

13.9.88 RANK (SOURCE).

Description. Returns the rank of an array or a scalar.

Kind. Inquiry function.

Argument. SOURCE may be of any type.

Result Type and Shape. Integer scalar.

Result Value. The result has value zero if SOURCE is scalar and otherwise has value equal to the rank of SOURCE.

Example. RANK ([1:N]) has the value one.

13.9.89 REAL (A, MOLD).

Optional Argument. MOLD

Description. Convert to real type.

Kind. Elemental function.

Arguments.

A	must be of type integer, real, double precision, or complex.
MOLD (optional)	must be of type real.

Result Type and Type Parameters. Real. If MOLD is present, the type parameters are those of MOLD; otherwise, they are the processor-dependent default type parameters for real type.

Result Value.

Case (i): If A is of type integer, real, or double precision, the result is equal to a processor-dependent approximation real part of A.

Case (ii): If A is of type complex, the result is equal to a processor-dependent approximation real part of A.

Example. REAL (−3) has the value −3.0.

13.9.90 REPEAT (STRING, NCOPIES).

Description. Concatenate several copies of a string.

Kind. Elemental function.

Arguments.

STRING	must be of type character.
NCOPIES	must be of type integer. Its value must not be negative.

Result Type and Type Parameters. Character of length NCOPIES times that of STRING.

Result Value. The value of the result is the concatenation of NCOPIES copies of STRING.

Example. REPEAT ('H', 2) has value 'HH'.

13.9.91 REPLICATE (ARRAY, DIM, NCOPIES).

Description. Replicates an array by increasing a dimension.

Kind. Transformational function.

Arguments.

ARRAY	may be of any type. It must not be scalar.
DIM	must be scalar and of type integer with value in the range $1 \le \text{DIM} \le n$, where n is the rank of ARRAY.
NCOPIES	must be scalar and of type integer.

Result Type, Type Parameters, and Shape. The result is an array of the same type, type parameters, and rank as ARRAY and has shape [E (1:DIM-1), MAX (NCOPIES, 0) ∗ E (DIM), E (DIM+1:n)], where the shape of ARRAY is E (1:n).

Result Value. Each element of the result has value equal to that of the corresponding element of ARRAY obtained by subtracting from subscript DIM sufficient integral multiples of E (DIM) to bring it into the range [1:E (DIM)].

Example. If A is the array $\begin{bmatrix} 2 & 3 \\ 3 & 4 \end{bmatrix}$, REPLICATE (A, DIM=2, NCOPIES=3) is $\begin{bmatrix} 2 & 3 & 2 & 3 & 2 & 3 \\ 3 & 4 & 3 & 4 & 3 & 4 \end{bmatrix}$.

13.9.92 RESHAPE (MOLD, SOURCE, PAD, ORDER).

Optional Arguments. PAD, ORDER

Description. Change the shape of an array.

Kind. Transformational function.

Arguments.

MOLD	must be of type integer and rank one. Its size must be positive and less than 8.
SOURCE	may be of any type. It must be array valued. Its shape must be defined. If PAD is absent, the size of SOURCE must be at least as great as that of the result.

PAD (optional)	must be of the same type and type parameters as SOURCE. PAD must be array valued.
ORDER (optional)	must be of type integer, must have the same shape as MOLD, and its value must be a permutation of [1:*n*], where *n* is the size of MOLD. If absent, it is as if it were present with value [1:*n*].

Result Type, Type Parameters, and Shape. The result is an array of shape MOLD (i.e., SHAPE (RESHAPE (MOLD, SOURCE)) = MOLD) with type and type parameters those of SOURCE.

Result Value. The elements of the result, taken in permuted subscript order ORDER (1), ..., ORDER (*n*), are those of SOURCE in normal subscript order value followed if necessary by those of PAD in subscript order value, followed if necessary by additional copies of PAD in subscript order value.

Example. RESHAPE ([2, 3], [1:6]) has value $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$.

13.9.93 RRSPACING (X).

Description. Returns the reciprocal of the relative spacing of model numbers near the argument value.

Kind. Elemental function.

Argument. X must be of type real.

Result Type and Type Parameters. Same as X.

Result Value. The result has value $|X \times b^{-e}| \times b^{p}$, where b, e, and p are as defined in 13.5.1 for the model representation of X, provided this result is within range.

Example. RRSPACING (−3.0) has the value 0.75×2^{24} for reals whose model is as at the end of 13.5.1.

13.9.94 SCALE (X, I).

Description. Returns $X \times b^{I}$ where b is the base in the model representation of X.

Kind. Elemental function.

Arguments.

X	must be of type real.
I	must be of type integer.

Result Type and Type Parameters. Same as X.

Result Value. The result has the value $X \times b^{I}$, where b is defined in 13.5.1 for model numbers representing values of X, provided this result is within range.

Example. SCALE (3.0, 2) has the value 12.0 for reals whose model is as at the end of 13.5.1.

13.9.95 SETEXPONENT (X, I).

Description. Returns the model number whose fractional part is the fractional part of the model representation of X and whose exponent part is I.

Kind. Elemental function.

Arguments.

X	must be of type real.
I	must be of type integer.

Result Type and Type Parameters. Same as X.

Result Value. The result has value X × b^{I-e}, where b and e are as defined in 13.5.1 for the model representation of X, provided this result is within range. If X has value zero, the result has value zero.

Example. SETEXPONENT (3.0, 1) has the value 1.5 for reals whose model is as at the end of 13.5.1.

13.9.96 SIGN (A, B).

Description. Absolute value of A times the sign of B.

Kind. Elemental function.

Arguments.

A	must be of type integer, real, or double precision.
B	must be of the same type as A.

Result Type and Type Parameters. Same as A.

Result Value. The value of the result is $|A|$ if $B \geq 0$ and $-|A|$ if $B < 0$.

Example. SIGN (−3.0, 2.0) has the value 3.0.

13.9.97 SIN (X).

Description. Sine function.

Kind. Elemental function.

Argument. X must be of type real, double precision, or complex.

Result Type and Type Parameters. Same as X.

Result Value. The result has value equal to a processor-dependent approximation to sin(X). If X is of type real or double precision, it is regarded as a value in radians. If X is of type complex, its real part is regarded as a value in radians.

Example. SIN (1.0) has the value 0.84147098 (approximately).

13.9.98 SINH (X).

Description. Hyperbolic sine function.

Kind. Elemental function.

Argument. X must be of type real or double precision.

Result Type and Type Parameters. Same as X.

Result Value. The result has value equal to a processor-dependent approximation to sinh(X).

Example. SINH (1.0) has the value 1.1752012 (approximately).

13.9.99 SPACING (X).

Description. Returns the absolute spacing of model numbers near the argument value.

Kind. Elemental function.

Argument. X must be of type real.

Result Type and Type Parameters. Same as X.

Result Value. The result has value b^{e-p}, where b, e, and p are as defined in 13.5.1 for the model representation of X, provided this result is within range; otherwise, the result is the same as that of TINY (X).

Example. SPACING (3.0) has the value 2^{-22} for reals whose model is as at the end of 13.5.1.

13.9.100 SPREAD (SOURCE, DIM, NCOPIES).

Description. Replicates an array by adding a dimension. Broadcasts several copies of SOURCE along a specified dimension (as in forming a book from copies of a single page) and thus forms an array of rank one greater.

Kind. Transformational function.

Arguments.

SOURCE	may be of any type. It may be scalar or array valued. The rank of SOURCE must be less than 7.
DIM	must be scalar and of type integer with value in the range $1 \leq \text{DIM} \leq n+1$, where n is the rank of SOURCE.
NCOPIES	must be scalar and of type integer.

Result Type, Type Parameters, and Shape. The result is an array of the same type and type parameters as SOURCE and of rank $n+1$, where n is the rank of SOURCE.

Case (i): If SOURCE is scalar, the shape of the result is [MAX (NCOPIES, 0)].

Case (ii): If SOURCE is array valued with shape E (1:n), the shape of the result is [E (1:DIM-1), MAX (NCOPIES, 0), E (DIM:n)].

Result Value.

Case (i): If SOURCE is scalar, each element of the result has value equal to SOURCE.

Case (ii): If SOURCE is array valued, the element of the result with subscript (r_1, r_2, ..., r_{n+1}) has the value SOURCE (s_1, s_2, ..., s_n) where (s_1, s_2, ..., s_n) is (r_1, r_2, ..., r_{n+1}) with subscript r_{DIM} omitted.

Example. If A is the array [2, 3, 4], SPREAD (A, DIM=1, NCOPIES=3) is the array $\begin{bmatrix} 2 & 3 & 4 \\ 2 & 3 & 4 \\ 2 & 3 & 4 \end{bmatrix}$.

13.9.101 SQRT (X).

Description. Square root.

Kind. Elemental function.

Argument. X must be of type real, double precision, or complex. Unless X is complex, its value must be greater than or equal to zero.

Result Type and Type Parameters. Same as X.

Result Value. The result has value equal to a processor-dependent approximation to the square root of X. A result of type complex 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.

Example. SQRT (4.0) has the value 2.0 (approximately).

13.9.102 SUM (ARRAY, DIM, MASK).

Optional Arguments. DIM, MASK

Description. Sum all the elements of ARRAY along dimension DIM with mask MASK.

Kind. Transformational function.

Arguments.

ARRAY	must be of type integer, real, double precision, or complex. It must not be scalar.
DIM (optional)	must be scalar and of type integer with value in the range $1 \le \text{DIM} \le n$, where n is the rank of ARRAY.
MASK	(optional) must be of type logical or bit and must be conformable with ARRAY.

Result Type, Type Parameters, and Shape. The result is of the same type and type parameters as ARRAY. It is scalar if DIM is absent or ARRAY has rank one; otherwise, the result is an array of rank $n-1$ and of shape $(d_1, d_2, ..., d_{\text{DIM}-1}, d_{\text{DIM}+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of ARRAY.

Result Value.

Case (i): The result of SUM (ARRAY) has value equal to a processor-dependent approximation to the sum of all the elements of ARRAY or has value zero if ARRAY has size zero.

Case (ii): The result of SUM (ARRAY, MASK) has value equal to a processor-dependent approximation to the sum of the elements of ARRAY corresponding to the true elements of MASK or has value zero if there are no true elements.

Case (iii): If ARRAY has rank one, SUM (ARRAY, DIM [,MASK]) has value equal to that of SUM (ARRAY [,MASK]). Otherwise, the value of element $(s_1, s_2, ..., s_{\text{DIM}-1}, s_{\text{DIM}+1}, ..., s_n)$ of SUM (ARRAY, DIM [,MASK]) is equal to SUM (ARRAY $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$ [, MASK $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$]).

Examples.

Case (i): The value of SUM ([1, 2, 3]) is 6.

Case (ii): SUM (C, MASK= C .GT. 0.0) forms the arithmetic sum of the positive elements of C.

Case (iii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$, SUM (B, DIM=1) is [3, 7, 11] and SUM (B, DIM=2) is [9, 12].

13.9.103 TAN (X).

Description. Tangent function.

Kind. Elemental function.

Argument. X must be of type real or double precision.

Result Type and Type Parameters. Same as X.

Result Value. The result has value equal to a processor-dependent approximation to tan(X), with X regarded as a value in radians.

Example. TAN (1.0) has the value 1.5574077 (approximately).

13.9.104 TANH (X).

Description. Hyperbolic tangent function.

Kind. Elemental function.

Argument. X must be of type real or double precision.

Result Type and Type Parameters. Same as X.

Result Value. The result has value equal to a processor-dependent approximation to tanh(X).

Example. TANH (1.0) has the value 0.76159416 (approximately).

13.9.105 TINY (X).

Description. Returns the smallest positive number in the model representing numbers of the same type and type parameters as the argument.

Kind. Inquiry function.

Argument. X must be of type real. It may be scalar or array valued.

Result Type, Type Parameters, and Shape. Scalar with the same type and type parameters as X.

Result Value. The result has value $b^{e_{\mathrm{min}}-1}$ where b and e_{min} are as defined in 13.5.1 for the model representing numbers of the same type and type parameters as X.

Example. TINY (X) has the value 2^{-127} for real X whose model is as at the end of 13.5.1.

13.9.106 TRANSPOSE (MATRIX).

Description. Transpose an array of rank two.

Kind. Transformational function.

Argument. MATRIX may be of any type and must have rank two. Its shape must be defined.

Result Type, Type Parameters, and Shape. The result is an array of the same type and type parameters as MATRIX and with rank two and shape [n, m] where [m, n] is the shape of MATRIX.

Result Value. Element (i, j) of the result has value MATRIX (j, i), $i = 1, 2, ..., n$; $j = 1, 2, ..., m$.

Example. If A is the array $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$, then TRANSPOSE (A) has the value

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}.$$

13.9.107 TRIM (STRING).

Description. Returns the argument with trailing blank characters removed.

Kind. Transformational function.

Argument. STRING must be of type character and must be a simple variable or array element (not an array or array section).

Result Type and Type Parameters. Character with a length that is the length of STRING less the number of trailing blanks in STRING.

Result Value. The value of the result is the same as STRING except any trailing blanks are removed. If STRING contains no nonblank characters, the result has zero length.

Example. TRIM(' A B ') has value ' A B'.

13.9.108 UNPACK (VECTOR, MASK, FIELD).

Description. Unpack an array of rank one into an array under the control of a mask.

Kind. Transformational function.

Arguments.

VECTOR	may be of any type. It must have rank one. Its size must be at least *t* where *t* is the number of true elements in MASK.
MASK	must be array valued and of type logical or bit. Its shape must be defined.
FIELD	must be of the same type and type parameters as VECTOR and must be conformable with MASK.

Result Type, Type Parameters, and Shape. The result is an array of the same type and type parameters as VECTOR and the same shape as MASK.

Result Value. The element of the result that corresponds to the i-th true element of MASK, counting in subscript order value, has value VECTOR (i) for $i = 1, 2, ..., t$, where t is the number of true values in MASK. Other elements have value equal to FIELD if FIELD is scalar or to the corresponding element of FIELD if it is an array.

Example. Specific values may be "scattered" to specific positions in an array by using UNPACK. If M is the array $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$, V is the array [1, 2, 3], and Q is the logical mask $\begin{bmatrix} . & T & . \\ T & . & . \\ . & . & T \end{bmatrix}$, where "T" represents .TRUE. and "." represents .FALSE., then the result of UNPACK (V, MASK=Q, FIELD=M) has the value $\begin{bmatrix} 0 & 2 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 3 \end{bmatrix}$ and the result of UNPACK (V, MASK=Q, FIELD=0) has the value $\begin{bmatrix} 0 & 2 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 3 \end{bmatrix}$.

13.9.109 VERIFY (STRING, SET).

Description. Verify that a set of characters contains all the characters in a string.

Kind. Elemental function.

Arguments.

STRING	must be of type character.
SET	must be of type character.

Result Type. Integer.

Result Value. The value of the result is zero if each character in STRING appears in SET or if STRING has zero length; otherwise, the value of the result is the position of the leftmost character of STRING that is not in SET.

Example. VERIFY ('AB', 'A') has value 2.

14 ENTITY SCOPE, ASSOCIATION, AND DEFINITION

An **entity** is one of the following:

(1) A program unit

(2) An internal procedure

(3) An interface block

(4) A common block

(5) An external input/output unit

(6) A statement function

(7) A type

(8) A simple variable

(9) A component

(10) A symbolic constant

(11) A statement label

(12) A construct

(13) A condition

14.1 Name and Scoping Rules.

14.1.1 Name of an Entity. All entities except statement labels and external input/output units have a name that is a **symbolic name**. A statement label is a string of one to five digits. An external input/output unit is referenced by an integer value.

14.1.1.1 Allowable Name Conflicts. In a program unit, two entities must not have the same name except as noted in the following paragraphs of this section. Two entities in different program units may have the same name except as noted in the following paragraphs of this section.

(1) A common block may have a name that is the same as the name of any local entity other than a symbolic constant, function, or a local variable that is the name of 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.

(2) 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 (12.5.2.2). At least one such function name must be the name of a variable in a function subprogram.

(3) A common block name may be the same as an array name, a structure name, a type name, a statement function name, or a component name.

(4) A component name in a program unit may also be the name of any local or global entity except another component name of the same type.

(5) A statement function dummy argument name may also be the name of a scalar 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 variable or common block. The statement function dummy argument name and variable name must have the same type and, if of type character, must have the same constant length.

(6) 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 variable or common block. The implied-DO-variable and the local variable have the same type.

(7) USE conflict (11.3.1)

14.1.2 Scope. The **scope** of an entity is that portion of an executable program in which a particular set of attributes is associated with the entity.

A **local entity** is one whose scope is less than an executable program.

14.1.2.1 Executable Program Scope. The following entities have a scope of an executable program.

(1) An external program unit

(2) A common block

(3) An external input/output unit

14.1.2.2 Program Unit Plus Keyword Scope. The following entities have a scope that is a program unit plus the lefthand portion of keyword arguments.

(1) A variable that is a dummy argument, except a statement function dummy argument.

(2) A procedure that is a dummy argument.

In the following example, X has this scope.

```
...
CALL A (X = P + Q)
...
SUBROUTINE A (X)
...
```

14.1.2.3 Program Unit Scope. The following entities have a scope that is a program unit.

(1) A variable that is not a dummy argument

(2) A procedure that is not a dummy argument

(3) A symbolic constant

(4) A statement function

(5) An intrinsic procedure

(6) An internal procedure

(7) A label

(8) A construct

14.1.2.4 Part of a Program Unit Scope. The following entities have a scope that is a program unit except for all contained internal procedures in which the entity is redeclared.

(1) ???

14.1.2.5 Statement Scope. The following entities have a scope that is a single statement.

(1) A variable that is the dummy argument of a statement function

(2) A DO variable in an implied-DO list of a DATA statement

(3) A *forall-var* in a FORALL statement (7.5.3)

14.2 Association.

Two entities may become associated by **name association** or by storage association.

14.2.1 Name Association. There are two forms of name association: **argument association** and **alias association**. Argument association provides a mechanism by which entities known in a procedure or main program unit (calling program) may be accessed in another procedure (called program). Alias association provides alternative avenues (e.g., different names) of access to an entity within a single program unit.

14.2.1.1 Argument Association. The rules governing argument association are given in Section 12. As explained in Section 12, execution of a procedure reference establishes an association between an actual argument and its corresponding dummy argument. Argument association may be either name based or storage sequence based. Storage sequence based argument association (14.2.2) occurs only when the actual argument is not conformable with an array dummy argument (e.g., when an array element is passed to an array dummy argument). All other instances of argument association are name based, in which argument attributes, rather than storage sequence, determine the nature of the association.

For name-based argument association of data objects, the actual and dummy arguments conform in type and shape. For procedures, the actual and dummy arguments must be the same kind (function or subroutine) of procedure, and for functions, the type and shape of the actual and dummy arguments must be conformable.

The name of the dummy argument may be different from the name, if any, of its associated actual argument. (Note that an actual argument may be a nameless data entity, such as an expression that is not simply a variable or constant.) The dummy argument name is the name by which the associated actual argument is known, and may be accessed by, in the called procedure.

Upon termination of execution of a procedure reference, all argument associations established by that reference are terminated. A dummy argument of that procedure may be associated with an entirely different actual argument in a subsequent execution of the procedure. That is, the scope of an argument association is the called procedure for the duration of the execution of the procedure reference that established the association.

14.2.1.2 Alias Association. An alias provides an alternative access to an entity within a single scope. The process of establishing an alias name for a given host or target entity, and the resulting relationship, is known as alias association. An alias association may be established by;

(1) Execution of an IDENTIFY statement or

(2) Renaming an entity imported with a USE statement.

14.2.1.2.1 Alias Association By IDENTIFY. The rules for the IDENTIFY form of name association are given in Section 6. Such an association may be applied only to variables. The alias name, if explicitly declared, must have the ALIAS attribute specified. Within the IDENTIFY statement the type and shape of the alias variable must conform to that of the host variable.

The scope of an identified association is the program unit in which the IDENTIFY statement occurs, and any procedures internal to that program unit. An alias association is established upon execution of an IDENTIFY statement and continues thereafter until the first occurrence of:

(1) Execution of another IDENTIFY statement in that program unit involving the same alias variable

(2) Termination of execution of the program unit

(3) Deallocation of the associated data object

Any variable (with conforming type and shape) may be the host for an alias, including allocated variables and other established aliases. An alias variable may not be defined prior to its association with a host variable. For an allocatable host, an alias association must not be established by execution of an IDENTIFY statement when the host is not allocated. Note that deallocation of a host variable terminates any alias associations with that variable.

Any number of alias variables may be concurrently associated with a given host. Each such alias provides access to the associated data object, and the host continues to be directly accessible by its original name (i.e., any associated alias may, but need not, be used to specify definition of or reference to the host variable). Any alias variable may be reassociated, by execution of IDENTIFY statements, any number of times to any set of (conformable) host variables during execution of the program unit.

14.2.1.2.2 Rename of Entities in a USE Statement. The rules for this form of alias association are given in 11.3.1. Such an association may be used to provide an alias name for a data object name (constant or variable) or a procedure name being made accessible by a USE statement (from either a module or host program unit).

A rename association has a scope of the program unit (including any internal procedures) containing the USE statement, and remains in effect throughout the execution of the executable program. If the program unit is a module subprogram, the scope of the alias name extends to program units containing a USE statement that references the module. Unlike identified alias name association, rename association cannot be changed or terminated during program execution. Rename aliases for allocatable variables may not be defined, nor made reference to, when the data object is not allocated, but, unlike identified aliases, rename alias associations are established prior to allocation and remain in effect after deallocation. Rename alias associations are permanent and static, whereas identified alias associations are temporary and dynamic.

A rename alias name must not appear in a type declaration, or otherwise have any attributes specified. It automatically assumes all attributes, and only those attributes, of its associated entity. In the scope of the association, the entity may be accessed only through its rename alias; the original name is not associated with the entity in this scope, and is available to be used for other purposes. A rename alias may be used in exactly the same way that the original name could have been used.

A rename alias must not appear in an IDENTIFY statement as an alias variable; it may appear in such a statement in a host variable context.

14.2.1.2.3 Summary Comparison of Identified and Rename Alias Associations.

Characteristic	Identified Associations	Rename Associations
Scope	Single program unit	Single program unit, plus using program units if in a module

Duration	Temporary	Entire program execution
May change?	Yes	No
How established?	Execution of IDENTIFY statement	Appearance in USE statement
How terminated?	Execution of IDENTIFY statement Deallocation of the entity Termination of execution of the executable program	Termination of execution of the executable program
Appearance in USE statement	Not allowed	Normal (only) way to establish
Appearance in IDENTIFY statement	As alias variable As host variable	as host variable
Allowed with unallocated host	No	Yes
May be allocated? (appear in ALLOCATE statement)	No	Yes
May be deallocated? (appear in DEALLOCATE statement)	Yes	Yes
Host name also accessible?	Yes	No
ALIAS attribute	Explicit or implicit for scalars, required for arrays, not allowed for procedures	Implicit for all entities

14.2.2 Storage Association. Storage sequences are used to describe relationships that exist among variables, array elements, substrings, common blocks, and arguments.

14.2.2.1 Storage Sequence. A **storage sequence** is a sequence 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 or a numeric storage unit.

A variable or array element of type integer, real, or logical has a storage sequence of one numeric storage unit.

A structure, structure component, bit data object, or structure element has no storage sequence.

A variable, array, or array element with explicitly specified precision and range attributes of type real or complex has no storage sequence.

A variable of type double precision or complex without explicitly specified precision and range 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 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 (4.3.2.1 and 5.1.1.3).

Each common block has a storage sequence (5.4.2.1).

Each data object appearing in a storage association context has a storage sequence (2.4.5).

14.2.2.2 Association of Storage Sequences. Two storage sequences s_1 and s_2 are **associated** if the ith storage unit of s_1 is the same as the jth storage unit of s_2. This causes the $(i + k)$th storage unit of s_1 to be the same as the $(j + k)$th storage unit of s_2, for each integer k such that $1 \le i + k \le$ ***size of*** s_1 and $1 \le j + k \le$ ***size of*** s_2.

14.2.2.3 Association of Data Objects. Two data objects are **storage 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 a data object affects the definition status and value of any associated entity. An EQUIVALENCE statement, a COMMON statement, an ENTRY statement, or a procedure reference may cause association of storage sequences.

An EQUIVALENCE statement causes association of data objects only within one program unit, unless one of the equivalenced entities is also in a common block (5.4.1.1 and 5.4.2.1).

COMMON statements cause data objects in one program unit to become associated with data object in another program unit.

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 only between two character entities or between a double precision or complex entity and an entity of type integer, real, logical, double precision, or complex.

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.

14.3 Definition and Undefinition. Certain events cause entities to become defined or become undefined.

14.3.1 Events That Cause Variables to Become Defined. Variables become defined as follows:

(1) Execution of an arithmetic, logical, structure, or character assignment statement causes the data object that precedes the equals to become defined.

(2) Execution of a masked array assignment causes some of the array elements in the array assignment block to become defined (7.5.2.2).

(3) Execution of an element array assignment (FORALL) statement causes some elements to become defined (7.5.3.2).

(4) As execution of an input statement proceeds, each entity that is assigned a value of its corresponding type from the input file becomes defined at the time of such assignment.

(5) Execution of a DO statement causes the DO-variable to become defined.

(6) Beginning of execution of the action specified by an implied-DO list in an input/output statement causes the implied-DO-variable to become defined.

(7) A DATA statement or a type declaration with an INITIAL attribute causes data objects to become initially defined at the beginning of execution of an executable program.

(8) Execution of an ASSIGN statement causes the variable in the statement to become defined with a statement label value.

(9) When a variable of a given type becomes defined, all totally associated data objects 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. When a variable of a given type becomes defined, all totally name associated variables of the same type become defined except that variables totally name associated with the variable in an ASSIGN statement become undefined when the ASSIGN statement is executed.

(10) 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.

(11) Execution of an input/output statement containing an input/output status specifier causes the specified integer variable or array element to become defined.

(12) Execution of an INQUIRE statement causes any variable that is assigned a value during the execution of the statement to become defined if no error condition exists.

(13) When a complex variable becomes defined, all partially associated real variables become defined.

(14) When both parts of a complex variable become defined as a result of partially associated real or complex variables becoming defined, the complex variable becomes defined.

(15) When all characters of a character variable become defined, the character variable becomes defined.

(16) When all components of a structure become defined, the structure becomes defined.

(17) When all elements of an array become defined, the array becomes defined.

(18) When all elements of an array section become defined, the array section becomes defined.

(19) Zero-sized arrays, zero-sized array sections, and zero-length substrings are always defined.

14.3.2 Events That Cause Variables to Become Undefined. Variables become undefined as follows:

(1) All variables are undefined at the beginning of execution of an executable program except zero-sized arrays, zero-length character variables, and those variables initially defined by DATA statements or a type declaration with an INITIAL attribute.

(2) When a variable of a given type becomes defined, all totally storage associated variables of different type become undefined.

(3) When a variable of a given type becomes defined, all totally name associated variables of different type become undefined.

(4) Execution of an ASSIGN statement causes the variable in the statement to become undefined as an integer. Variables that are associated with the variable become undefined.

(5) When a variable of type other than character becomes defined, all partially associated variables become undefined. However, when a variable of type real is partially associated with a variable of type complex, the complex variable does not become undefined when the real variable becomes defined and the real variable does not become undefined when the complex variable becomes defined. When an variable of type complex is partially associated with another variable of type complex, definition of one variable does not cause the other to become undefined.

(6) When the evaluation of a function causes an argument of the function or a variable 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 variable in common becomes undefined when the expression is evaluated .

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

- (a) Entities specified by SAVE statements
- (b) Entities in blank common
- (c) Entities in a named common block that appears in the subprogram and appears in at least one other program unit that is making either a direct or indirect reference to the subprogram
- (d) Entities whose scope has been extended from the containing program unit, if there is one
- (e) Entities whose scope has been extended from a module subprogram that is also the source for a USE statement in a program unit that makes a direct or indirect reference to a subprogram containing the RETURN statement or END statement.

(8) When an error condition or end-of-file condition occurs during execution of an input statement, all of the variables specified by the input list of the statement become undefined.

(9) Execution of a direct access input statement that specifies a record that has not been previously written causes all of the variables specified by the input list of the statement to become undefined.

(10) Execution of an INQUIRE statement may cause variables to become undefined (9.6).

(11) When any character of a character variable becomes undefined, the character variable becomes undefined.

(12) When a variable becomes undefined as a result of conditions described in (6) through (11), all totally associated variables become undefined and all partially associated variables of type other than character become undefined.

(13) When a variable becomes undefined as a result of conditions described in (5) through (10), all totally name associated variables become undefined and all partially associated variables of type other than character become undefined.

(14) When any component of a structure and any other component containing that component becomes undefined, the structure becomes undefined. This does not imply that the undefinition of one component of a structure causes all other components to become undefined. Redefinition or undefinition of the tag name component also causes undefinition of components selected by all cases.

(15) When an array element becomes undefined, the array and any array sections containing that array element become undefined. This does not imply that the undefinition of one array element causes any other array element to become undefined.

(16) The signalling of conditions causes entities to become undefined.

15 DEPRECATED FEATURES

In this standard, the deprecated features are identified by a distinguishing type font (1.5.1) and are intended to be removed from the next version of the Fortran standard. These deprecated features are a part of the current language described in this standard. In this section, the deprecated features are listed. Possible replacements are indicated in Appendix B.

The deprecated features are:

(1) Assumed-size dummy arrays

(2) An array element associated with a dummy argument array

(3) BLOCK DATA subprogram

(4) COMMON statement

(5) EQUIVALENCE statement

(6) ENTRY statement

(7) Fixed source form

(8) Specific names for intrinsic functions

(9) Statement function

(10) Arithmetic IF statement

(11) Computed GO TO statement

(12) DATA statement

(13) DIMENSION statement

(14) Double precision data type

(15) Alternate return

(16) ASSIGN and assigned GO TO statement

(17) PAUSE statement

(18) Real and double precision DO variables

APPENDIX A FORTRAN FAMILY OF STANDARDS

A host language standard, such as Fortran, should take responsibility for coordinating other standards built on its base to prevent the development of conflicting collateral standards. A Fortran Reference Model has been suggested for the **Fortran Family of Standards.**

The Fortran Family of Standards consists of:

(1) The Fortran Language Standard

(2) Supplementary Standards based on Procedure Libraries

(3) Supplementary Standards based on Module Libraries

(4) Secondary Standards

X3.9-1978 (the previous Fortran standard) is referred to as **Fortran 77** in this appendix. X3.9-198x is referred to as **Fortran 8x**. the next standard is referred to as **Fortran 9x**.

A.1 The Fortran Language Standard. The Fortran Language consists of **primary features** from Fortran 77, **decremental features** that are deprecated in this standard and marked for possible deletion in the next Fortran standard, and **incremental features** that add new constructs to the Fortran standard. (See Figure 1.)

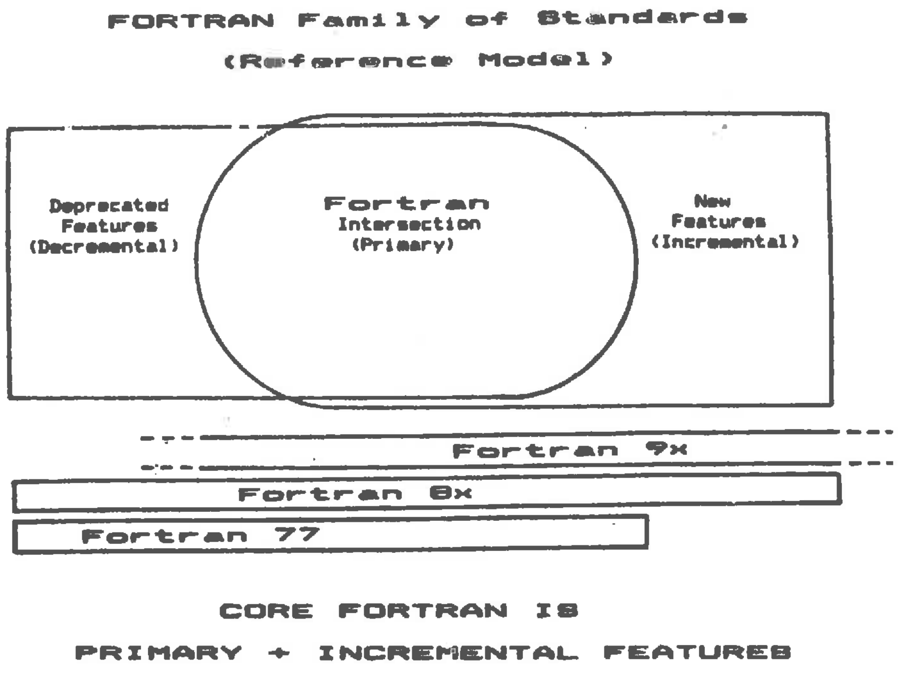
FORTRAN Family of Standards
(Reference Model)
Deprecated Features (Decremental)
Fortran Intersection (Primary)
New Features (Incremental)
Fortran 9x
Fortran 8x
Fortran 77
CORE FORTRAN IS
PRIMARY + INCREMENTAL FEATURES

Figure 1. The Fortran Language Standard.

A.1.1 Primary Features. These features are those from the Fortran 77 standard that continue to be useful and characteristic of the language. Primary features are expected to continue throughout the life of Fortran or at least for the next several revisions of the language.

A.3 Supplementary Standards Based on Module Libraries. A supplementary standard based on module subprograms is called a **module supplementary standard**. Supplementary standards may specify module subprograms that provide a high level of application-oriented functionality. These may include the defining of new data types and their accompanying operators. Module subprograms are nonexecutable program units containing definitions made available to any other program unit by the USE statement. Many problem-oriented applications would make excellent candidates for module supplementary standards. Modules may be included in the Fortran Standard document or they may be standardized in separate documents.

A.3.1 Interface Mechanisms. The interface mechanisms provided in Fortran 8x contain a set of facilities for binding a variety of additional features, such as graphics, to Fortran. These facilities include module subprograms which make definitions, data declarations, and procedure libraries available to an executable program. The USE statement provides the means for referencing specific module subprograms. Supplementary standards may use these mechanisms in defining a specific process within the Fortran Family of Standards.

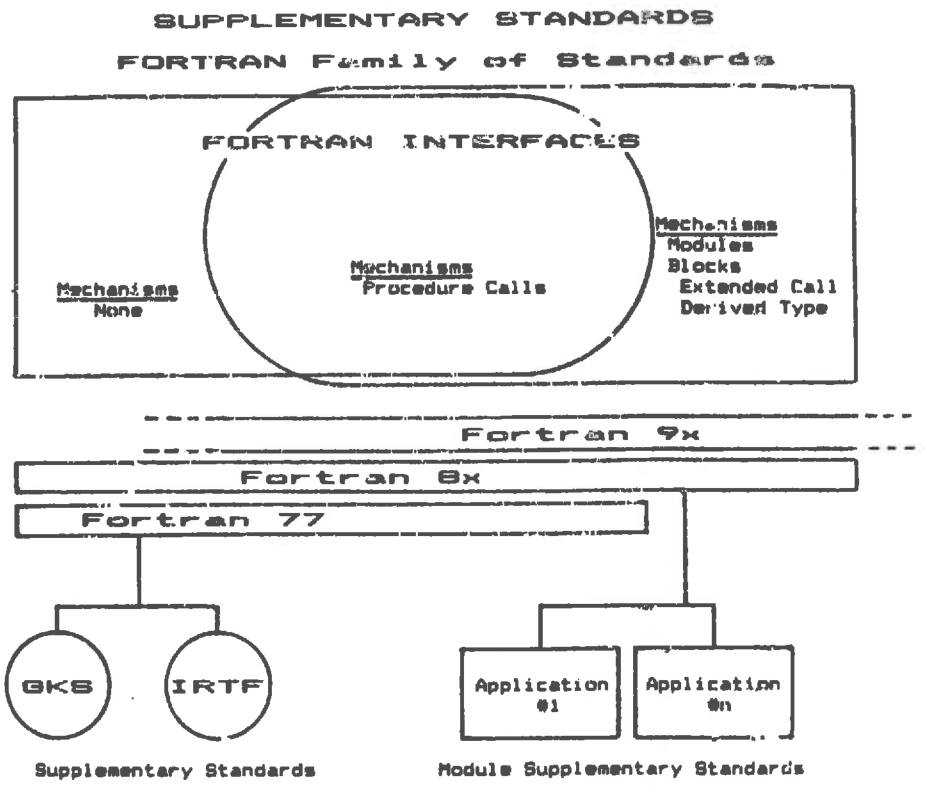

Figure 2. Supplementary Standards.

A.3.2 Rules. Some rules governing the preparation of supplementary standards that are based on procedure and module libraries are:

(1) A module may be appended to the Fortran Standard or it may be a separate standard.

(2) If a module is appended to the Fortran Standard, it is forwarded for review at the same time as the standard. If it is a separate supplementary standard, there is an independent standardization process.

(3) A module is not part of the Standard. It is a member of the Fortran Family of Standards.

(4) Standard modules must not use deprecated features (i.e., must conform to the Fortran Core.) When the Fortran Standard is revised, a formerly standard-conforming module may become not standard conforming because of the use of (new) decremental features.

(5) When the Fortran Standard is revised, a review may determine that modifications are needed to take advantage of any new functionality (incremental features) in the standard.

(6) A name registration for supplementary standards is available from the Fortran Standards Technical Subcommittee.

(7) Separate standards projects should be defined (SD-3) for each supplementary and secondary standard. Task groups may be formed within the Fortran Standards Technical Subcommittee for development of supplementary and secondary standards.

(8) Standard Modules prepared outside the committee and its task groups must use the interface mechanisms in the language. Requests for new facilities in the Fortran Standard must be processed by the Fortran Standards Technical Subcommittee.

(9) The Fortran Standards Technical Subcommittee should review all candidates for supplementary and secondary standards to determine if they are standard conforming. This must be done in a timely manner.

A.4 Secondary Standards. Secondary standards do not impact or change the syntax of the language nor do they change the semantics of the Fortran Standard. Instead, these standards may make requirements on the conformance of programs using the Fortran Standard. For example, certain constructs that control the execution sequence of a program may be required to flag specific conditions that occur during execution. Validation of programs during compilation or execution is another example. Conformance requirements could be expanded in a separate secondary standard. The syntax rules used to help describe the form that Fortran statements take are included in the Fortran Standard (1.5). These rules are described in a variation of BNF. A formal grammar might also be produced as a separate document. Currently, there are no secondary standards in the Fortran Family of Standards; however, work is proceeding in these areas for Fortran and for programming languages in general. See Figure 3.

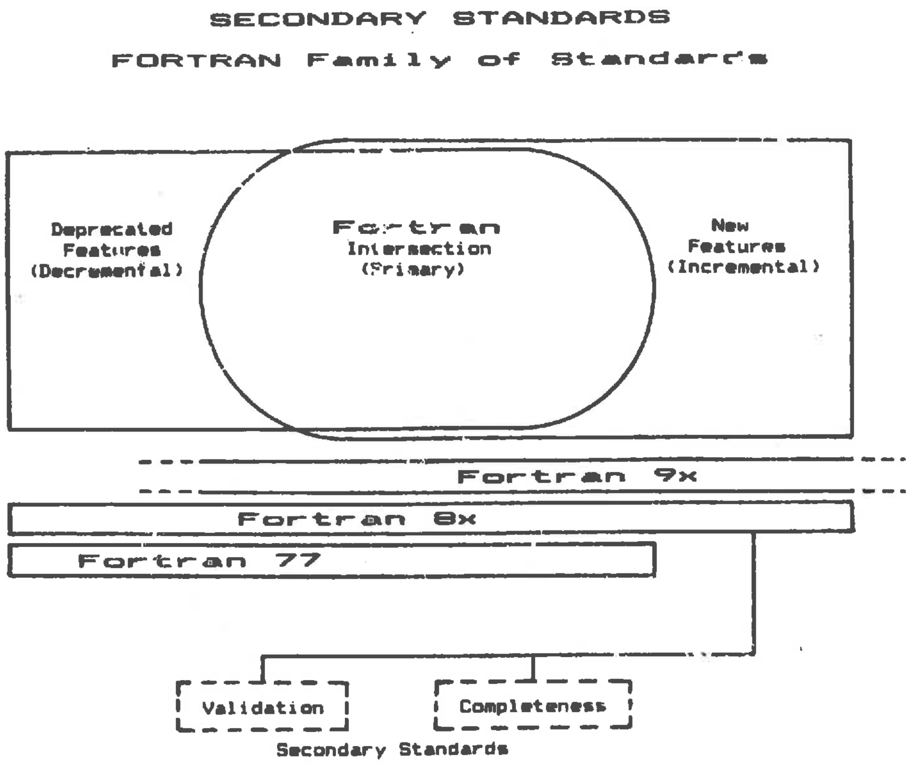
SECONDARY STANDARDS
FORTRAN Family of Standards
Deprecated Features (Decremental)
Fortran Intersection (Primary)
New Features (Incremental)
Fortran 9x
Fortran 8x
Fortran 77
Validation
Completeness
Secondary Standards

Figure 3. Secondary Standards.

A.5 Standard Conformance. Any program unit containing syntax not defined in the Fortran language is not standard conforming with respect to the Fortran Standard. The inclusion of a USE statement does not make the nonstandard conforming syntax standard conforming. A program unit that uses only syntax and semantics defined in the Fortran language standard and one or more standard modules is standard conforming with respect to the Fortran Family of Standards.

In moving to a revised standard, a number of features rather than the complete standard are often selected by implementors. It is recommended that partial implementations of major features not be done. For example, if the array facilities are to be included, as many of the array features as possible should be implemented.

A.5.1 Name Registration. A list of names registered with the Fortran Standards Technical Subcommittee will be kept for reference by those who are preparing a module intended for the Fortran Family of Standards.

A.6 Fortran Family of Standards. Figure 4 is the complete diagram of the Fortran Family of Standards. It includes the Fortran language with incremental, decremental, and primary features. The interface mechanisms shown refer to the procedure and module supplementary standards in the reference model.

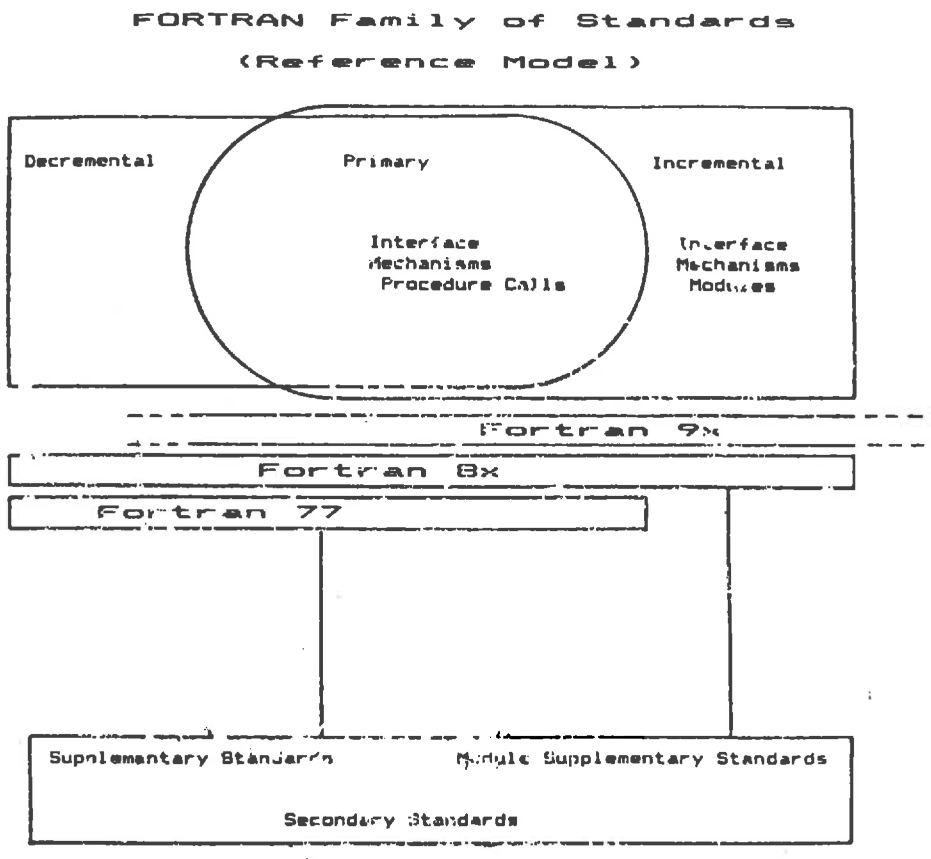
FORTRAN Family of Standards
(Reference Model)
Decremental
Primary
Incremental
Interface Mechanisms Procedure Calls
Interface Mechanisms Modules
Fortran 9x
Fortran 8x
Fortran 77
Supplementary Standards
Module Supplementary Standards
Secondary Standards

Figure 4. The Fortran Family of Standards

APPENDIX B DEPRECATED FEATURES NOTES

This appendix more fully describes the rationale for the specific deprecated features. Possible alternatives to the deprecated features are described.

B.1 Storage Association. Storage association is the association of data objects through storage sequence patterns rather than by object identification. Storage association allows the user to configure regions of storage and to conserve the use of storage by dynamically designating the objects contained within these storage regions. Though the disadvantages of the use of storage association have been known for some time, features added in this standard have provided Fortran with adequate replacement facilities for important functionality formerly only provided by storage association. The six items below are deprecated due to their use of storage association.

B.1.1 Assumed-Size Dummy Arrays. These are dummy arrays declared using an asterisk to specify its last dimension. In this standard, dummy arrays may be declared as assumed-shape arrays by using the colon with no upper bound in one or more dimension positions of the dummy array declaration. Assumed-shape arrays include all of the functionality of assumed-size arrays. Assumed-size arrays assume that a contiguous set of array elements is being passed. With assumed-shape arrays, an array section that does not consist of a contiguous set of array elements (such as a row of a matrix) may also be passed.

B.1.2 Passing an Array Element or Substring to a Dummy Array. This functionality is now achieved more safely by passing the desired array section. For example, if a one-dimensional array XX is to be passed starting with the sixth element, then instead of passing XX (6) to the dummy array, one would pass the array section XX (6:); if the eleventh through forty-fifth elements are to be passed, the actual argument is the array section XX (11:45).

B.1.3 BLOCK DATA Subprogram. The principal use of BLOCK DATA subprograms is to initialize common blocks. Modules provide a complete replacement for BLOCK DATA subprograms. The global data functionality of common blocks is also provided by modules. Global data in modules may be initialized when specified.

B.1.4 COMMON Statement. The important functionality of the COMMON statement has been in its use in specifying global data pools. In this standard, global data pools may be provided more safely and conveniently with MODULE program units and USE statements. Using the COMMON statement, a global data pool could be specified by:

```
INTEGER X (1000)
REAL Y (100, 100)
COMMON / POOL1 / X, Y
```

Each program unit using this global data would need to contain these specifications. Alternatively, one can define the global data pool in a MODULE program unit:

```
MODULE POOL1
   INTEGER X (1000)
   REAL Y (100, 100)
END MODULE
```

Each program unit using this global data would contain the statement

```
USE POOL1
```

When used in this manner, the MODULE/USE functionality is similar to the INCLUDE extension in many Fortran implementations. This is safer than using common blocks because the specification of the global data pool appears only once. In addition, the USE statement is very short and easy to use. Facilities are provided in the USE statement (not shown here) to rename module objects if different names are desired in the program unit using the module objects.

Another advantage is that modules do not involve storage association. Therefore, they may contain any desired mix of character, noncharacter, and structured objects. Because a common block involves storage association, a common block cannot contain both character and noncharacter data objects.

B.1.5 ENTRY Statement. The ENTRY statement is typically used in situations where there are several operations involving the same set of data objects:

```
procedure-heading
  data-specifications
entry_1
  ...
  RETURN
entry_2
  ...
  RETURN
...
entry_n
  ...
  RETURN
END
```

The MODULE program unit provides the equivalent functionality in the form:

```
MODULE module-name
  data-specifications
procedure_1
  ...
  END
procedure_2
  ...
  END
...
procedure_n
  ...
  END
END MODULE
```

A program unit using this module may call each procedure in it, exactly as if they were entry points. One advantage is that some of the procedures in a module may be functions and some may be subroutines, whereas all entry points in a function procedure must be invoked as functions and all entry points in a subroutine procedure must be invoked as subroutines.

B.1.6 EQUIVALENCE Statement. A major use of the EQUIVALENCE statement is to have two or more data objects, possibly of different types, share the same storage region. This was important in earlier periods when address space was limited making conservation necessary. The EQUIVALENCE statement also provides the means of simulating certain data types, structures, and transfer functions. This functionality is now available in the language.

Reuse of storage can now be achieved by using automatic arrays (5.1.2.4.1) and allocatable arrays. Following the return from the subprogram, the space for the dynamic local array is available for reuse.

The derived type capability provides a replacement for the more awkward means of achieving data structures through the use of EQUIVALENCE statements.

The ability of the EQUIVALENCE statement to alias two or more data objects or remap two or more arrays is now provided by the IDENTIFY statement.

B.2 Redundant Functionality. The eight features identified below are deprecated simply because they are now completely redundant, having been superseded.

(1) Fixed source form — replaced by the new source form (3.3)

(2) Specific names for intrinsic function — use generic names (13.1)

(3) Statement functions — replaced by internal functions (12.1.2.2)

(4) Arithmetic-IF statement — replaced by logical-IF and block-IF (8.1.2)

(5) Computed GOTO statement — replaced by SELECT construct (see B.2.2 below)

(6) DATA statement — replaced by INITIAL specifier (5.1.2.1.2), and by assignment of array constants (Section 7)

(7) DIMENSION statement — use type declaration instead (5.1)

(8) DOUBLE PRECISION statement — use precision control attributes (4.3.1.2, 5.1.1.3)

(9) Shared DO termination and termination on a statement other than END DO — use an END DO statement for each DO statement

B.2.1 Use of Internal Functions for Statement Functions. The functionality of the internal function provides a better replacement for the limited statement function capability. For example:

function-name (*dummy-arguments*) = *expr*

may be replaced by the internal function definition:

FUNCTION *function-name* (*dummy-arguments*)
function-name = *expr*
END

The use of an internal function in a program unit is the same as the use of a statement function.

B.2.2 Example Replacement of the Computed GO TO Statement. The execution sequence controlled by the computed GO TO:

GO TO (*label*$_1$, *label*$_2$, ..., *label*$_n$), *integer-variable*
  ...
  GO TO *label*$_z$
label$_1$ CONTINUE
  ...
  GO TO *label*$_z$
label$_2$ CONTINUE
  ...
  GO TO *label*$_z$

```
...
label_n CONTINUE
  ...
  GO TO label_z
label_z CONTINUE
```

may be replaced by the SELECT CASE construct:

```
SELECT CASE (integer-variable)
     CASE DEFAULT
       ...
     CASE (1)
       ...
     CASE (2)
       ...
     CASE (n)
       ...
END SELECT
```

Also see Section 8.1.3.

B.2.3 Alternate RETURN. Alternate returns introduce labels into an argument list to allow the called program to direct the execution sequence of the called subprogram upon return. Readability and maintainability suffer when alternate returns are used. A better practice is to provide a return code argument that is set by the called subprogram and used in a SELECT CASE construct of the calling program unit to direct its subsequent execution. Maintainability is enhanced because an additional SELECT CASE construct may be added without modifying the actual and dummy argument lists.

```
      CALL subr-name (X, Y, Z, *100, *200, *300, ...)
      ...
100   CONTINUE
      ...
      GO TO 999
200   CONTINUE
      ...
      GO TO 999
      ...
999   CONTINUE
```

where labels 100, 200, 300, etc., are alternate return points. In many cases, the effect can be more safely achieved with a return code and a SELECT CASE structure:

```
CALL subr-name (X, Y, Z, RETURN_CODE)
SELECT CASE (RETURN_CODE)
  CASE (return_1)
    ...
  CASE (return_2)
    ...
  ...
END SELECT
```

B.2.4 ASSIGN and Assigned GO TO. The ASSIGN statement allows a label to be dynamically assigned to an integer variable, and the assigned GO TO statement allows "indirect branching" through this variable. This hinders the readability of the program flow, especially if the integer variable also is used in arithmetic operations. The two totally different usages of the integer variable can be an obscure source of error.

Previously, internal subroutines were simulated by the presence of remote code blocks in a procedure. The ASSIGN GOTO provided the simulated return from the remote code block "internal subroutine". The addition of internal subroutines to the language replaced this error prone usage.

Example:

```
ASSIGN 120 TO RETURN      ! SET UP RETURN POINT
GOTO 740                  ! BRANCH TO "SUBROUTINE"
120 CONTINUE
...
740 CONTINUE
...                       ! "SUBROUTINE" BODY
GOTO RETURN               ! "SUBROUTINE" RETURN
...
```

This functionality also is provided in this standard through the use of internal subroutines:

```
CALL SUBR_740
...
SUBROUTINE SUBR_740
...                   ! SUBROUTINE BODY
END
...
```

This illustrates the use of internal subroutines to conveniently provide "remote code block" functionality.

B.2.5 PAUSE Statement. Execution of a PAUSE statement requires operator or system-specific intervention to resume execution. In most cases, the same functionality can be achieved as effectively and in a more portable way with the use of an appropriate READ statement that awaits some input data.

B.3 Redundant Functionality. A number of features are deprecated because they are now completely redundant. Redundant features and their possible replacements, using features in the core, are given in Table B.2.

Table B.2.

Deprecated Feature	Core Alternative
Fixed source form	Free source form
Specific names for intrinsic functions	Generic names
Statement functions	Internal functions
Arithmetic IF statement	IF construct or IF statement
Computed GO TO	SELECT construct

DATA statement	INITIAL attribute or INITIALIZE statement
DIMENSION statement	Type declaration
DOUBLE PRECISION statement	Specified REAL precision
Non-END-DO DO termination	END DO statement
Shared DO termination	END DO statement for each DO loop

APPENDIX C SECTION NOTES

C.1 Section 1 Notes. Use of deprecated features is discouraged. Each deprecated feature may be considered for removal in the next revision of the Fortran standard.

C.2 Section 2 Notes. Keywords can make procedure references more readable and allow actual arguments to be in any order. This latter property permits optional arguments.

C.3 Section 3 Notes. A partial collating sequence is specified. If possible, a processor should use the American National Standard Code for Information Interchange, ANSI X3.4-1977 (ASCII), sequence for the complete Fortran character set.

The standard does not restrict the number of consecutive comment lines. The limit of 19 continuation lines or 1320 characters permitted for a statement should not be construed as being a limitation on the number of consecutive comment lines.

There are 99999 unique statement labels and a processor must accept 99999 as a statement label. However, a processor may have an implementation limit on the total number of unique statement labels in one program unit.

Blanks are not permitted within statement labels in free source form.

C.4 Section 4 Notes. A processor must not consider a negative zero to be different from a positive zero.

C.5 Section 5 Notes.

C.6 Section 6 Notes.

C.7 Section 7 Notes. The Fortran 77 restriction that none of the character positions being defined in the character assignment statement may be referenced in the expression has been removed (7.5.1.5).

C.8 Section 8 Notes.

C.9 Section 9 Notes. What is called a "record" in Fortran is commonly called a "logical record". There is no concept in Fortran of a "physical record".

An endfile record does not necessarily have any physical embodiment. The processor may use a record count or other means to register the position of the file at the time an ENDFILE statement is executed, so that it can take appropriate action when that position is reached again during a read operation. The endfile record, however it is implemented, is considered to exist for the BACKSPACE statement.

This standard accommodates, but does not require, file cataloging. To do this, several concepts are introduced.

Before any input/output can be performed on a file, it must be connected to a unit. The unit then serves as a designator for that file as long as it is connected. To be connected does not imply that "buffers" have or have not been allocated, that "file-control tables" have or have not been filled out, or that any other method of implementation has been used.

Connection means that (barring some other fault) a READ or WRITE statement can be executed on the unit, hence on the file. Without a connection, a READ or WRITE statement cannot be executed.

Totally independent of the connection state is the property of existence, this being a file property. The processor "knows" of a set of files that exist at a given time for a given executable program. This set would include tapes ready to read, files in a catalog, a keyboard, a printer, etc. The set may exclude files inaccessible to the executable program because of security, because they are already in use by another executable program, etc. This standard does not specify which files exist, hence wide latitude is available to a processor to implement security, locks, privilege techniques, etc. Existence is a convenient concept to designate all of the files that an executable program can potentially process.

All four combinations of connection and existence may occur:

Connect	Exist	Examples
Yes	Yes	A card reader loaded and ready to be read
Yes	No	A printer before the first line is written
No	Yes	A file named 'JOAN' in the catalog
No	No	A reel of tape destroyed in the fire last week

Means are provided to create, delete, connect, and disconnect files.

A file may have a name. The form of a file name is not specified. If a system does not have some form of cataloging or tape labeling for a least some of its files, all file names will disappear at the termination of execution. This is a valid implementation. Nowhere does this standard require names to survive for any period of time longer than the execution time span of an executable program. Therefore, this standard does not impose cataloging as a prerequisite. The naming feature is intended to allow use of a cataloging system where one exists.

A file may become connected to a unit in either or two ways: preconnection or execution of an OPEN statement. Preconnection is performed prior to the beginning of execution of an executable program by means external to Fortran. For example, it may be done by job control action or by processor established defaults. Execution of an OPEN statement is not required to access preconnected files.

The OPEN statement provides a means to access existing files that are not preconnected. An OPEN statement may be used in either of two ways: with a file name (open by name) and without a file name (open by unit). A unit is given in either case. Open by name connects the specified file to the specified unit. Open by unit connects a processor-determined default file to the specified unit. (The default file may or may not have a name.)

Therefore, there are three ways a file may become connected and hence processed: preconnection, open by name, and open by unit. Once a file is connected, there is no means in standard Fortran to determine how it became connected.

An OPEN statement may also be used to create a new file. In fact, any of the foregoing three connection methods may be performed on a file that does not exist. When a unit is preconnected, writing the first record created the file. With the other two methods, execution of the OPEN statement creates the file.

When a unit becomes connected to a file, either by execution of an OPEN statement or by preconnection, the following connection properties may be established:

(1) An access method, which is sequential or direct, is established for the connection.

(2) A form, which is formatted or unformatted, is established for a connection to a file that exists or is created by the connection. for a connection that results from execution of an OPEN statement, a default form (which depends on the access method, as described in 9.2.1.2) is established if no form is specified. For a preconnected file that exists, a form is established by preconnection. For a preconnected file that does not exist, a form may be established, or the establishment of a form may be delayed until the file is created (for example, by execution of a formatted or unformatted WRITE statement).

(3) A record length may be established. If the access method is direct, the connection established a record length, which specifies the length of each record of the file. A connection for sequential access does not have this property.

(4) A blank significance property, which is ZERO or NULL, is established for a connection for which the form is formatted. This property has no effect on output. For a connection that results from execution of an OPEN statement, the blank significance property is NULL by default if no blank significance property is specified. For a preconnected file, the property is established by preconnection.

The blank significance property of the connection is effective at the beginning of each formatted input statement. During execution of the statement, any BN or BZ edit descriptors encountered may temporarily change the effect of embedded and trailing blanks.

A processor has wide latitude in adapting these concepts and actions to its own cataloging and job control conventions. Some processors may require job control action to specify the set of files that exist or that will be created by an executable program. Some processors may require no job control action prior to execution. This standard enables processors to perform a dynamic open, close, and file creation, but it does not require such capabilities of the processor.

The meaning of “open” in contexts other that Fortran may include such things as mounting a tape, console messages, spooling, label checking, security checking, etc. These actions may occur upon job control action external to Fortran, upon execution of an OPEN statement, or upon execution of the first read or write of the file. The OPEN statement describes properties of the connection to the file and may or may not cause physical activities to take place. It is a place for an implementation to define properties of a file beyond those required in standard Fortran.

Similarly, the actions of dismounting a tape, protection, etc. of a “close” may be implicit at the end of a run. The CLOSE statement may or may not cause such actions to occur. This is another place to extend file properties beyond those of standard Fortran. Note, however, that the execution of a CLOSE statement on unit 10 followed by an OPEN statement on the same unit to the same file or to a different file is a permissible sequence of events. The processor must not deny this sequence solely because the implementation chooses to do the physical act of closing the file at the termination of execution of the program.

This standard does not address problems of security, protection, locking, and many other concepts that may be part of the concept of “right of access”. Such concepts are considered to be in the province of an operating system.

The OPEN and INQUIRE statements can be extended naturally to consider these things.

Possible access methods for a file are: sequential and direct. The processor may implement two different types of files, each with its own access method. It may also implement one type of file with two different access methods.

Direct access to files is of a simple and commonly available type, that is, fixed-length records. The key is a positive integer.

Keyword forms of specifiers are used because there are many specifiers and a positional notation is difficult to remember. The keyword form sets a style for processor extensions. The UNIT= and FMT= keywords are offered for completeness, but their use is optional. Thus, compatibility with ANSI X3.9-1966 and ANSI X3.9-1978 is achieved.

If no list items are specified in a list-directed input/output statement, one input record is skipped or one empty output record is written.

An example of a restriction on input/output statements (9.8) is that an input statement must not specify that data are to be read from a printer.

If a character constant is used as a format specifier in an input/output statement, care must be taken that the value of the character constant is a valid format specification. In particular, if the format specification contains an apostrophe edit descriptor, two apostrophes must be written to delimit the apostrophe edit descriptor and four apostrophes must be written for each apostrophe that occurs within the apostrophe edit descriptor. For example, the text:

```
2 ISN'T 3
```

may be written by various combinations of output statements and format specifications:

```
    WRITE (6, 100) 2, 3
100 FORMAT (1X, I1, 'ISN''T', 1X, I1)

    WRITE (6, '(1X, I1, 1X, ''ISN''''T'', 1X, I1)') 2, 3

    WRITE (6, '(A)') ' 2 ISN''T 3'
```

The T edit descriptor includes the carriage control character in lines that are to be printed. T1 specifies the carriage control character and T2 specifies the first character that is printed.

The length of a record is not always specified exactly and may be processor dependent.

The number of records read by a formatted input statement can be determined from the following rule: A record is read at the beginning of the format scan (even if the input list is empty), at each slash edit descriptor encountered in the format, and when a format rescan occurs at the end of the format.

The number of records written by a formatted output statement can be determined from the following rule: A record is written when a slash edit descriptor is encountered in the format, when a format rescan occurs at the end of the format, and at completion of execution of the output statement (even if the output list is empty). Thus, the occurrence of n successive slashes between two other edit descriptors causes $n - 1$ blank lines if the records are printed. The occurrence of n slashes at the beginning or end of a complete format specification causes n blank lines if the records are printed. However, a complete format specification containing n slashes ($n > 0$) and no other edit descriptors causes $n + 1$ blank lines if the records are printed. For example, the statements

```
  PRINT 3
3 FORMAT (/)
```

will write two records that cause two blank lines if the records are printed.

The following examples illustrate list-directed input. A blank character is represented by b.

Example 1:

Program:

```
J = 3
READ *, I
READ *, J
```

Sequential input file;

```
b1b,4bbbbb
,2bbbbbbbb
```

Result: I = 1, J = 3.

Explanation: The second READ statement reads the second record. The initial comma in the record designates a null value; therefore, J is not redefined.

Example 2:

Program:

```
CHARACTER A *8, B *1
READ *, A, B
```

Sequential input file:

Result: A = 'bbbbbbbb', B = 'Q'

Explanation: The end of a record cannot occur between two apostrophes representing an embedded apostrophe in a character constant; therefore, A is set to the character constant 'bbbbbbbb'. The end of a record acts as a blank, which in this case is a value separator because it occurs between two constants.

C.10 Section 10 Notes.

C.11 Section 11 Notes. Program units that do not reference modules are independent of any other program unit. That is, their definitions are completely self-contained and they may be processed without having access to any other program unit. Program units that reference modules, however, are dependent upon the referenced modules, and processing such program units requires access to the definitions contained in the referenced modules. The manner in which this access is provided is implementation dependent, though it is sufficient to have the referenced module source text available during processing of the referencing program unit. The manner in which objects in a module are made accessible to the executable program is implementation dependent.

The following example of a module defines a rather complete data abstraction for a SET data type where the elements of each set are of type integer. The standard set operations of UNION (+), INTERSECTION (*), and DIFFERENCE (−) are provided. The CARD function returns the cardinality of (number of elements in) its set argument. Two functions returning logical values are included, ELEMENT and SUBSET, both of which have the operator form .IN.; ELEMENT determines if a given scalar integer value is an element of a given set, and SUBSET determines if a given set is a subset of another given set. (Two sets may be checked for equality by comparing cardinality and checking that one is a subset of the other, or checking to see if each is a subset of the other.)

The transfer function SET converts a vector of integer values to the corresponding set, with duplicate values removed. Thus, a vector of constant values can be used as set constants. An inverse transfer function VECTOR returns the elements of a set as a vector of values in ascending order. An assignment coercion allows assignment between sets of different sizes, and checks to see if the receiving set data object has an adequate maximum size (returning the null set if not). In this SET implementation, set data objects have a maximum size (number of elements in set) of 200.

Examples (A, B, and C are sets; X is an integer variable):

```
IF (CARD(A) .GT. 10) ...       ! CHECK TO SEE IF A HAS MORE THAN 10 ELEMENTS

IF (X .IN. A .AND. .NOT. X .IN. B) ... ! CHECK FOR X AN ELEMENT OF A BUT NOT OF B

C = A + (B * SET (1:100))      !  C IS THE UNION OF A AND THE
                               !  RESULT OF B INTERSECTED WITH THE INTEGERS 1 TO 100

IF (CARD (A * SET (2:100:2)) .GT. 0) ... ! DOES A HAVE ANY EVEN
                                         ! NUMBERS IN THE RANGE 1:100?

PRINT *, VECTOR (B)            !  PRINT OUT THE ELEMENTS OF SET B, IN ASCENDING ORDER

MODULE INTEGER_SETS

IMPLICIT  TYPE SET ( A-I, U ),  INTEGER (X)

TYPE SET                                   ! DEFINE SET DATA TYPE
     INTEGER CARDINAL_NUMBER
     INTEGER ELEMENT_VALUE(200)            ! COULD BE ANY DATA TYPE
END TYPE SET

INTEGER FUNCTION CARD (A)                  ! RETURNS CARDINALITY OF SET A
     CARD = A % CARDINAL_NUMBER
END FUNCTION CARD

LOGICAL FUNCTION ELEMENT (X,A) OPERATOR (.IN.)    ! DETERMINES IF
     ELEMENT = .FALSE.                            ! ELEMENT X IS IN SET A
     IF (CARD(A) .EQ. 0) RETURN
     IF (ANY (A % ELEMENT_VALUE (1:CARD(A)) .EQ. X))  ELEMENT = .TRUE.
END FUNCTION ELEMENT

FUNCTION UNION (A,B) OPERATOR (+)          ! UNION BETWEEN SETS A AND B
     N = CARD (A)
     UNION = SET (A % ELEMENT_VALUE(1:N))
     DO J=1, CARD (B)
        IF (.NOT. B % ELEMENT_VALUE(J) .IN. A) THEN
                 N = N+1
                 UNION % ELEMENT_VALUE(N) = B % ELEMENT_VALUE (J)
        END IF
     END DO
     UNION % CARDINAL_NUMBER = N
END FUNCTION UNION

FUNCTION DIFFERENCE (A,B) OPERATOR (-)     ! DIFFERENCE OF SETS A AND B
     DIFFERENCE = SET ([1:0])
     DO J=1, CARD(A)
          X = A % ELEMENT_VALUE(J)
          IF (.NOT. (X .IN. B)) DIFFERENCE = DIFFERENCE + SET(X)
     END DO
END FUNCTION DIFFERENCE

FUNCTION INTERSECTION (A,B) OPERATOR (*)    ! INTERSECTION OF SETS A AND B
     INTERSECTION = A - (A-B)
```

```
END FUNCTION INTERSECTION

LOGICAL FUNCTION SUBSET (A,B) OPERATOR (.IN.)  ! DETERMINES IF SET A IS A
     LOGICAL L (SIZE(A % ELEMENT_VALUE))       ! SUBSET OF SET B -
     SUBSET = CARD (A) .LE. CARD (B)! OVERLOADS .IN. OPERATION
     IF (.NOT. SUBSET) RETURN
     FOR ALL ( J=1:CARD(A) ) L(J) = A % ELEMENT_VALUE(J) .IN. B
     IF (.NOT. ALL(L)) SUBSET = .FALSE.
END FUNCTION SUBSET

FUNCTION SET(V)                              ! TRANSFER FUNCTION BETWEEN A
     INTEGER V(:)                            ! CORRESPONDING SET OF ELEMENTS
     SET % CARDINAL_NUMBER = 0               ! REMOVING DUPLICATE VALUES
     DO J=1,SIZE(V)
         IF (.NOT. V(J).IN.SET) THEN
                  SET % CARDINAL_NUMBER = SET % CARDINAL_NUMBER + 1
                  SET % ELEMENT_VALUE (SET % CARDINAL NUMBER) = V(J)
         END IF
     END DO
END FUNCTION SET

FUNCTION VECTOR (A)                          ! TRANSFER THE VALUES OF SET A
     INTEGER VECTOR(:)                       ! INTO A VECTOR OF ASCENDING ORDER
     INTEGER I
     ALLOCATE ( VECTOR(CARD(A)) )
     VECTOR = A % ELEMENT_VALUE (1:CARD(A))
     DO I=1,CARD(A)-1
         DO J=1,CARD(A)-I
             IF (VECTOR(J+1) .LT. VECTOR(J) THEN
                K = VECTOR(J); VECTOR(J) = VECTOR(J+1); VECTOR(J+1) = K
             END IF
         END DO
     END DO
END FUNCTION VECTOR

SUBROUTINE SET_ASSIGNMENT_COERCION (A,B) ASSIGNMENT
     A = SET( ); N = CARD(B)
     IF (SIZE (A % ELEMENT_VALUE) .GE. N) A = SET (B % ELEMENT_VALUE(1:N))
END SUBROUTINE SET_ASSIGNMENT_COERCION

END MODULE INTEGER_SETS
```

C.12 Section 12 Notes. In 12.5.2.8, the intention of rules (2), (3), and (4) is to permit a variety of implementations, including the association of actual and dummy arguments by location or by copy-in/copy-out.

C.13 Section 13 Notes.

C.13.1 Summary of Features. This section is a summary of the principal array features.

C.13.1.1 Whole Array Expressions and Assignments. An important extension is that whole array expressions and assignments will be permitted. For example, the statement

```
A = B + C * SIN (D)
```

where A, B, C, and D are arrays of the same shape, is permitted. It is interpreted element-by-element; that is, the sine function is taken on each element of D, each result is multiplied by the corresponding element of C, added to the corresponding element of B, and assigned to the corresponding element of A. Functions, including user-written functions, may be array valued and may overload scalar versions having the same name. All arrays in an expression or across an assignment must "conform"; that is, have exactly the same "rank" (number of dimensions) and "shape" (set of lengths in each dimension), but scalars may be included freely and these are interpreted as being broadcast to a conforming array. Expressions are evaluated before any assignment takes place.

C.13.1.2 Array Sections. Whenever whole arrays may be used, it is also possible to use rectangular slices called "sections". For example:

```
A(:, 1:N, 2, 3:1:-1)
```

consists of a subarray containing the whole of the first dimension, positions 1 to N of the second dimension, position 2 of the third dimension and positions 1 to 3 in reverse order for the fourth dimension. This is an artificial example chosen to illustrate the different forms. Of course, the most common use is to select a row or column of an array, for example:

```
A (:, J)
```

C.13.1.3 WHERE and FORALL Statements. There are two mechanisms for restricting an array assignment. The WHERE statement applies a conforming logical array as a mask on the individual operations in the expression and in the assignment. For example:

```
WHERE (A .GT. 0) B = LOG (A)
```

takes the logarithm only for positive components of A and makes assignments only in these positions.

The FORALL statement is used whenever it is convenient or necessary to have access to the actual subscript values in an array expression and assignment. For example:

```
FORALL (I = 1:N, J = 1:N, I .GE. J) L(I, J) = A(I, J)
```

performs an array assignment over the lower triangular part of the leading $N \times N$ submatrix.

The WHERE statement also has a block form (WHERE construct).

C.13.1.4 Automatic and Allocatable Arrays. A major advance for writing modular software will be the presence of AUTOMATIC arrays, created on entry to a subprogram and destroyed on return, and ALLOCATABLE arrays whose rank is fixed but whose actual size and lifetime is fully under the programmer's control through explicit ALLOCATE and DEALLOCATE statements. The declarations

```
SUBROUTINE X (N, A, B)
REAL WORK (N, N), HEAP (:, :)
```

are associated with an automatic array WORK and an allocatable array HEAP. Note that a stack is an adequate storage mechanism for the implementation of automatic arrays, but a heap will be needed for allocatable arrays.

C.13.1.5 Array Constructors. Arrays, and in particular array constants, may be constructed with array constructors exemplified by:

```
[1.0, 3.0, 7.2]
```

which is an array of size 3,

```
[10[1.3,2.7], 7.1]
```

which has size 21 and contains [1.3,2.7] repeated 10 times followed by 7.1, and

```
[1:N]
```

which contains the integers 1, 2, ..., N. Only rank-one arrays may be constructed in this way, but higher dimensional arrays may be made from them by means of the intrinsic function RESHAPE.

C.13.1.6 The IDENTIFY Statement. At its simplest, the IDENTIFY statement permits the construction of subarrays that do not lie along the axes. As a simple example:

```
IDENTIFY (DIAG (I) = A (I, I), I = 1 : N)
```

constructs a vector that overlays the main diagonal of A. After execution of such an IDENTIFY statement, the alias array DIAG so constructed can be used whenever an array of the same shape might be used.

C.13.1.7 Intrinsic Functions. All of the Fortran 77 intrinsic functions and all of the scalar intrinsic functions that have been added to the language have been extended to be applicable to arrays. The function is applied element-by-element to produce an array of the same shape. In addition, the following array intrinsics have been added, many of which return array-valued results.

C.13.1.7.1 Vector and Matrix Multiply Functions.

DOTPRODUCT(VECTOR_A,VECTOR_B)	Dot product of two arrays
MATMUL(MATRIX_A,MATRIX-B)	Matrix multiplication

C.13.1.7.2 Array Reduction Functions.

ALL(ARRAY,DIM)	True if all values are true
ANY(ARRAY,DIM)	True if any value is true
COUNT(ARRAY,DIM)	Number of true elements in an array.
MAXVAL(ARRAY,DIM,MASK)	Maximum value in an array
MINVAL(ARRAY,DIM,MASK)	Minimum value in an array
PRODUCT(ARRAY,DIM,MASK)	Product of array elements
SUM(ARRAY,DIM,MASK)	Sum of array elements

C.13.1.7.3 Array Inquiry Functions.

ALLOCATED(ARRAY)	Space allocation
LBOUND(ARRAY,DIM)	Lower dimension bounds of an array
RANK(SOURCE)	Rank of an array or scalar
SHAPE(ARRAY)	Shape of an array
SIZE(ARRAY,DIM)	Total number of array elements
UBOUND(ARRAY,DIM)	Upper dimension bounds of an array

C.13.1.7.4 Array Construction Functions.

`DIAGONAL(VECTOR,FILL)`	Create a diagonal matrix
`MERGE(TSOURCE,FSOURCE,MASK)`	Merge under mask
`PACK(ARRAY,MASK,VECTOR)`	Pack array into a vector under a mask
`REPLICATE(ARRAY,DIM,NCOPIES)`	Replicates an array by increasing a dimension
`RESHAPE(MOLD,SOURCE,PAD,ORDER)`	Reshape an array
`SPREAD(SOURCE,DIM,NCOPIES)`	Replicates an array by adding a dimension
`UNPACK(VECTOR,MASK,FIELD)`	Unpack a vector into an array under a mask

C.13.1.7.5 Array Manipulation Functions.

`CSHIFT(ARRAY,DIM,SHIFT)`	Circular shift
`EOSHIFT(ARRAY,DIM,SHIFT,BOUNDARY)`	End-off shift
`TRANSPOSE(MATRIX)`	Transpose of matrix

C.13.1.7.6 Array Geometric Functions.

`FIRSTLOC(MASK,DIM)`	Locate first true element
`LASTLOC(MASK,DIM)`	Locate last true element
`PROJECT(ARRAY,MASK,BACKGROUND,DIM)`	Select masked values

C.13.2 Examples. The array features have the potential to simplify the way that almost any array-using program is conceived and written. Many algorithms involving arrays can now be written conveniently as a series of computations with whole arrays.

C.13.2.1 Unconditional Array Computations. At the simplest level statements such as A=B+C or S=SUM(A) can take the place of entire DO loops. The loops were required to do array addition or to sum all the elements of an array.

Further examples of unconditional operations on arrays that are simple to write are:

matrix multiply	`P = MATMUL(Q,R)`
largest array element	`L = MAXVAL(P)`
factorial N	`F = PRODUCT([2:N])`

The Fourier sum $F = \sum_{i=1}^{N} a_i \times \cos x_i$ may also be computed without writing a DO loop if one makes use of the element-by-element definition of array expressions as described in Section 7. Thus, we can write

```
F = SUM (A * COS (X)).
```

The successive stages of calculation of F would then involve the arrays:

```
        A   =   [A(1),...,A(N)]
        X   =   [X(1),...,X(N)]
   COS(X)   =   [COS(X(1)),...,COS(X(N))]
 A*COS(X)   =   [A(1)*COS(X(1)),...,A(N)*COS(X(N))]
```

The final scalar result is obtained simply by summing the elements of the last of these arrays. Thus, the compiler is dealing with arrays at every step of the calculation.

C.13.2.2 Conditional Array Computations. Suppose we wish to compute the Fourier sum in the above example, but to include only those terms $a(i)\cos x(i)$ that satisfy the condition that the coefficient $a(i)$ is less than 0.01 in absolute value. More precisely, we are now interested in evaluating the conditional Fourier sum

$$CF = \sum_{|a_i| < 0.01} a_i \times \cos x_i$$

where the index runs from 1 to N as before.

This can be done using the MASK parameter of the SUM function, which restricts the summation of the elements of the array A * COS(X) to those elements that correspond to true elements of MASK. Clearly, the logical expression required as the mask is ABS(A) .LT. 0.01. Note that the stages of evaluation of this expression are:

```
              A  =  [A(1),...,A(N)]
         ABS(A)  =  [ABS(A(1)),...,ABS(A(N))]
ABS(A) .LT. 0.01  =  [ABS(A(1) .LT. 0.01,...,ABS(A(N)) .LT. 0.01]
```

The conditional Fourier sum we arrive at is:

```
CF = SUM (A * COS (X), MASK = ABS (A) .LT. 0.01)
```

If the mask is all false, the value of CF is zero.

The use of a mask to define a subset of an array is crucial to the action of the WHERE statement. Thus for example, to set an entire array to zero, we may write simply A = 0; but to set only the negative elements to zero, we need to write the conditional assignment

```
WHERE (A .LT. 0)  A = 0
```

The WHERE statement complements ordinary array assignment by providing array assignment to any subset of an array that can be restricted by a logical expression.

In the Ising model described below, the WHERE statement predominates in use over the ordinary array assignment statement.

C.13.2.3 A Simple Program: The Ising Model. The Ising model is a well-known Monte Carlo simulation in 3-dimensional Euclidean space which is useful in certain physical studies. We will consider in some detail how this might be programmed. The model may be described in terms of a logical array of shape N by N. Each gridpoint is a single logical variable which is to be interpreted as either an up-spin (true) or a down-spin (false).

The Ising model operates by passing through many successive states. The transition to the next state is governed by a local probabilistic process. At each transition, all gridpoints change state simultaneously. Every spin either flips to its opposite state or not according to a rule that depends only on the states of its 6 nearest neighbors in the surrounding grid. The neighbors of gridpoints on the boundary faces of the model cube are defined by assuming cubic periodicity. In effect, this extends the grid periodically by replicating it in all directions throughout space.

The rule states that a spin is flipped to its opposite parity for certain at points where a mere 3 or fewer of the 6 nearest neighbors currently have the same parity as it does. Also, the flip is executed only with probability P(4), P(5), or P(6) if as many as 4, 5, or 6 of them have the same parity as it does. (The rule seems to promote neighborhood alignments that may presumably lead to equilibrium in the long run).

C.13.2.3.1 Problems To Be Solved. Some of the programming problems that we will need to solve in order to translate the Ising model into Fortran statements using entire arrays are:

(1) Counting nearest neighbors that have the same spin;

(2) Providing an array-valued function to return an array of random numbers; and

(3) Determining which gridpoints are to be flipped.

C.13.2.3.2 Solutions In Fortran. The arrays needed are:

```
LOGICAL ISING (N, N, N), FLIPS (N, N, N)
INTEGER ONES (N, N, N), COUNT (N, N, N)
REAL RANDOTHRESHOLD (N, N, N)
```

The array-valued function needed is:

```
FUNCTION RANDOM (N, N, N)
REAL THRESHOLD (N, N, N)
```

The transition probabilities may be passed across in the array

```
REAL P(6)
```

The first task is to count the number of nearest neighbors of each gridpoint *g* that have the same spin as *g*.

Assuming that ISING is given to us, the statements

```
ONES = 0
WHERE (ISING) ONES = 1
```

makes the array ONES into an exact analog of ISING in which 1 stands for an up-spin and 0 for a down-spin.

The next array we construct, COUNT, will record for every gridpoint of ISING the number of spins to be found among the 6 nearest neighbors of that gridpoint. COUNT will be computed by adding together 6 arrays, one for each of the 6 relative positions in which a nearest neighbor is found. Each of the 6 arrays is obtained from the ONES array by shifting the ONES array one place circularly along one of its dimensions. This use of circular shifting imparts the cubic periodicity.

```
COUNT = CSHIFT(ONES, DIM = 1, SHIFT = -1)  &
       +CSHIFT(ONES, DIM = 1, SHIFT =  1)  &
       +CSHIFT(ONES, DIM = 2, SHIFT = -1)  &
       +CSHIFT(ONES, DIM = 2, SHIFT =  1)  &
       +CSHIFT(ONES, DIM = 3, SHIFT = -1)  &
       +CSHIFT(ONES, DIM = 3, SHIFT =  1)
```

At this point, COUNT contains the count of nearest neighbor up-spins even at the gridpoints where the Ising model has a down-spin. But we want a count of down-spins at those gridpoints, so we correct COUNT at the down (false) points of ISING by writing:

```
WHERE (.NOT. ISING)  COUNT = 6 - COUNT
```

Our object now is to use these counts of what may be called the "like-minded nearest neighbors" to decide which gridpoints are to be flipped. This decision will be recorded as the true elements of an array FLIP. The decision to flip will be based on the use of uniformly distributed random numbers from the interval $0 \le p \le 1$. These will be provided at each gridpoint by the array-valued function RANDOM. The flip will occur at a given point if and only if the random number at that point is less than a certain threshold value. In particular, by making the threshold value equal to 1 at the points where there are 3 or fewer like-minded nearest neighbors, we guarantee that a flip occurs at those points (because p is always less than 1). Similarly, the threshold values corresponding to counts of 4, 5, and 6 are set to P(4), P(5), and P(6) in order to achieve the desired probabilities of a flip at those points (P(4), P(5), and P(6) are input parameters in the range 0 to 1).

The thresholds are established by the statements:

```
THRESHOLD = 1
WHERE (COUNT .EQ. 4) THRESHOLD = P(4)
WHERE (COUNT .EQ. 5) THRESHOLD = P(5)
WHERE (COUNT .EQ. 6) THRESHOLD = P(6)
```

and the spins that are to be flipped are located by the statement:

```
FLIPS = RANDOM (N) .LE. THRESHOLD
```

All that remains to complete one transition to the next state of the ISING model is to reverse the spins in ISING wherever FLIPS is true:

```
WHERE (FLIPS)  ISING = .NOT. ISING
```

C.13.2.3.3 The Complete Fortran Subroutine. The complete code, enclosed in a subroutine that performs a sequence of transitions, is as follows:

```
SUBROUTINE TRANSITION (N, ISING, ITERATIONS, P)

LOGICAL ISING (N, N, N), FLIPS (N, N, N)
INTEGER ONES (N, N, N), COUNT (N, N, N)
REAL THRESHOLD (N, N, N), P (6)

!     This interface block is needed to specify
!     that RANDOM is array-valued.
INTERFACE
    FUNCTION RANDOM (N)
    REAL RANDOM (N, N, N)
END INTERFACE

DO (ITER = 1, ITERATIONS)
   ONES = 0
   WHERE (ISING)  ONES = 1
   COUNT = CSHIFT (ONES, 1, -1) + CSHIFT (ONES, 1, 1) &
          +CSHIFT (ONES, 2, -1) + CSHIFT (ONES, 2, 1) &
          +CSHIFT (ONES, 3, -1) + CSHIFT (ONES, 3, 1)
   WHERE (.NOT. ISING)  COUNT = 6 - COUNT
   THRESHOLD = 1
   WHERE (COUNT .EQ. 4)  THRESHOLD = P(4)
   WHERE (COUNT .EQ. 5)  THRESHOLD = P(5)
   WHERE (COUNT .EQ. 6)  THRESHOLD = P(6)
   FLIPS = RANDOM (N) .LE. THRESHOLD
   WHERE (FLIPS)  ISING = .NOT. ISING
END DO
END
```

C.13.2.3.4 Reduction of Storage. The array ISING could be removed (at some loss of clarity) by representing the model in ONES all the time. The array FLIPS can be avoided by combining the two statements that use it as:

```
WHERE (RANDOM (N) .LE. THRESHOLD)  ISING = .NOT. ISING
```

but an extra temporary array would probably be needed. Thus, the scope for saving storage while performing whole array operations is limited. If N is small, this will not matter and the use of whole array operations is likely to lead to good execution speed. If N is large, storage may be very important and adequate efficiency will probably be available by performing the operations plane by plane. The resulting code is not as elegant, but all the arrays except ISING will have size of order N^2 instead of N^3.

C.13.3 FORmula TRANslation and Array Processing. Many mathematical formulas can be translated directly into Fortran by use of the array processing features.

We assume the following array declarations:

```
REAL X (N), A (M, N)
```

Some examples of mathematical formulas and corresponding Fortran expressions follow.

C.13.3.1 A Sum of Products. The expression

$$\sum_{j=1} Pi \overset{M}{=} 1\, a_{ij}$$

can be formed using the Fortran expression

```
SUM (PRODUCT (A, DIM=1))
```

The argument DIM = 1 means that the product is to be computed down each column of A. If A had the value

$$\begin{bmatrix} A & B & C \\ D & E & F \end{bmatrix}$$

the result of this expression is AD + BE + CF.

C.13.3.2 A Product of Sums. The expression

$$\underset{i=1}{\overset{M}{P}} \sum_{j=1}^{N} a_{ij}$$

can be formed using the Fortran expression

```
PRODUCT (SUM (A, DIM = 2))
```

The argument DIM = 2 means that the sum is to be computed along each row of A. If A had the value

$$\begin{bmatrix} A & B & C \\ D & E & F \end{bmatrix}$$

the result of this expression is (A + B + C)(D + E + F).

C.13.3.3 Addition of Selected Elements. The expression

$$\sum_{x_i > 0.1} x_i$$

can be formed using the Fortran expression

```
SUM (X, MASK = X .GT. 0.1)
```

The mask locates the elements where the array of rank one is greater than 0.1. If X had the value [0.0, 0.1, 0.2, 0.3, 0.2, 0.1, 0.0], the result of this expression is 0.7.

C.13.4 Variance from the Mean. The expression

$$\sum_{i=1}^{N} (x_i - x_{\text{mean}})^2$$

can be formed using the Fortran statements

```
XMEAN = SUM (X) / SIZE (X)
VAR = SUM ((X - XMEAN) ** 2)
```

Thus, VAR is the sum of the squared residuals.

C.13.5 Matrix Norms: Euclidean Norm.

```
NORM2 (A) = SQRT (SUM (A**2))
```

(Note: The Euclidean norm of a real matrix is the square root of the sum of the squares of its elements)

C.13.6 Matrix Norms: Maximum Norm.

```
NORM_INFINITY (A) = MAXVAL (SUM (ABS (A), DIM = 2))
```

(Note: The maximum_norm of a real matrix is the largest row_ sum of the matrix of its absolute values)

C.13.7 Logical Queries. The intrinsic functions allow quite complicated questions about tabular data to be answered without use of loops or conditional constructs. Consider, for example, the questions asked below about a simple tabulation of students' test scores.

Suppose the rectangular table T (M, N) contains the test scores of M students who have taken N different tests. T is an integer matrix with entries in the range 0 to 100. Example A: the scores on 4 tests made by 3 students held as the table

$$T = \begin{bmatrix} 85 & 76 & 90 & 60 \\ 71 & 45 & 50 & 80 \\ 66 & 45 & 21 & 55 \end{bmatrix}$$

Question: What is each student's top score?

Answer: MAXVAL (T, DIM = 2); in Example A: [90, 80, 66].

Question: What is the average of all the scores?

Answer: SUM (T) / SIZE (T); in Example A: 62.

Question: How many of the scores in the table are above average?

Answer: ABOVE = T .GT. SUM (T) / SIZE (T); N = COUNT (ABOVE); in Example A: ABOVE is the logical array (t = true, . = false):

$$\begin{bmatrix} t & t & t & . \\ t & . & . & t \\ t & . & . & . \end{bmatrix}$$

and COUNT (ABOVE) is 6.

Question: What was the lowest score in the above-average group of scores?

Answer: MINVAL (T, MASK = ABOVE), where ABOVE is as defined previously; in Example A: 66.

Question: Was there a student whose scores were all above average?

Answer: With ABOVE as previously defined the answer is yes or no according as the value of the expression ANY (ALL (ABOVE, DIM = 2)) is true or false; in Example A the answer is no.

C.13.8 Parallel Computations. The most straightforward kind of parallel processing is to do the same thing at the same time to many operands. Matrix addition is a good example of this very simple form of parallel processing. Thus, the array assignment A = B + C specifies that corresponding elements of the identically-shaped arrays B and C be added together in parallel and that the resulting sums be assigned in parallel to the array A.

The "process" being done "in parallel" in the example of matrix addition is of course the process of addition. And the array feature that so successfully implements matrix addition as a parallel process is the element-by-element evaluation of array expressions.

These observations lead us to look to element-by-element computation as a means of implementing other simple parallel processing algorithms.

The applications of element-by-element computation to parallel processing include the following:

C.13.8.1 Parallel Evaluation of Polynomials. This encompasses both the evaluation of several polynomials at one point and the evaluation of one polynomial at several points.

C.13.8.2 Parallel Computation of FFTs. In radar signal processing it is convenient to perform the Fast Fourier Transform in parallel on many sets of radar signals (each such set might consist of, say, 64 complex numbers).

C.13.8.3 Parallel Sorting. We will address the problem of sorting the several columns of a matrix in parallel.

C.13.8.4 Parallel Finite Differencing. We will examine the parallel computation of finite difference approximations to partial derivatives at all points of a grid.

C.13.9 Examples of Element-by-Element Computation.

C.13.9.1 Polynomials. Several polynomials of the same degree may be evaluated at the same point by arranging their coefficients as the rows of a matrix and applying Horner's method for polynomial evaluation to the COLUMNS of the matrix so formed.

This procedure is illustrated by the code to evaluate the three cubic polynomials:
in parallel at the point t = X and to place the
resulting vector of numbers [P(X), Q(X), R(X)] in
the real array RESULT (3).

The code to compute RESULT is just the one statement

```
RESULT = M (:, 1) + X * ( M (:, 2) + X * ( M (:, 3) + X * M (:, 4)))
```

where M represents the matrix M (3, 4) with value

$$P(t) = 1 + 2t - 3t^2 + 4t^3$$

$$Q(t) = 2 - 3t + 4t^2 - 5t^3$$

$$R(t) = 3 + 4t - 5t^2 + 6t^3$$

$$40 \quad \begin{bmatrix} 1 & 2 & -3 & 4 \\ 2 & -3 & 4 & -5 \\ 3 & 4 & -5 & 6 \end{bmatrix}$$

C.14 Section 14 Notes.

C.15 Section 15

APPENDIX D SYNTAX RULES

1 INTRODUCTION

2 FORTRAN TERMS AND CONCEPTS

R201	*executable-program*	**is**	*external-program-unit*
			[*external-program-unit*]...

Constraint: An *executable-program* must contain exactly one *main-program* *program-unit*.

R202	*external-program-unit*	**is**	*main-program*
		or	*external-subprogram*
R203	*main-program*	**is**	[*program-stmt*]
			program-unit-body
			end-program-stmt
R204	*external-subprogram*	**is**	*external-proc-subprogram*
		or	*module-subprogram*
		or	*block-data-subprogram*
R205	*external-proc-subprogram*	**is**	*procedure-subprogram*
R206	*procedure-subprogram*	**is**	*function-subprogram*
		or	*subroutine-subprogram*
R207	*function-subprogram*	**is**	*function-stmt*
			program-unit-body
			end-function-stmt
R208	*subroutine-subprogram*	**is**	*subroutine-stmt*
			program-unit-body
			end-subroutine-stmt
R209	*module-subprogram*	**is**	*module-stmt*
			program-unit-body
			end-module-stmt

Constraint: A module *program-unit-body* must not contain an *execution-part*.

R210	*block-data-subprogram*	is	*block-data-stmt*

Constraint: A *block-data-subprogram* *program-unit-body* may contain only IMPLICIT, PARAMETER, type declaration, COMMON, DIMENSION, EQUIVALENCE, DATA, and SAVE statements.

R211	*program-unit-body*	**is**	[*use-stmt*]...
			[*implicit-part*]...
			[*declaration-part*]...
			[*stmt-function-part*]...
			[*execution-part*]...
			[*contains-stmt*
			[*internal-proc-subprogram*]...]

R212 *implicit-part* **is** *implicit-stmt*
or *parameter-stmt*
or *format-stmt*
or *entry-stmt*

Constraint: The last *implicit-part*, if any, in a program unit body must be an *implicit-stmt*.

R213 *declaration-part* **is** *derived-type-def*
or *interface-block*
or *type-declaration-stmt*
or *specification-stmt*
or *parameter-stmt*
or *format-stmt*
or *entry-stmt*

R214 *stmt-function-part* **is** *format-stmt*
or *data-stmt*
or *entry-stmt*
or *stmt-function-stmt*

Constraint: The first *stmt-function-part*, if any, in a program unit body must be a *stmt-function-stmt*.

R215 *execution-part* **is** *executable-construct*
or *format-stmt*
or *data-stmt*
or *entry-stmt*

Constraint: The first *execution-part*, if any, in a program unit body must be an *executable-construct* or a DATA statement.

R216 *internal-proc-subprogram* **is** *proc-subprogram*

R217 *specification-stmt* **is** *access-stmt*
or *condition-stmt*
or *exponent-letter-stmt*
or *external-stmt*
or *initialize-stmt*
or *intent-stmt*
or *intrinsic-stmt*
or *optional-stmt*
or *range-stmt*
or *save-stmt*
or *common-stmt*
or *dimension-stmt*
or *equivalence-stmt*

Constraint: An *intent-stmt* or *optional-stmt* must not appear in a module or main program because they apply only to dummy arguments.

R218 *executable-construct* **is** *action-stmt*
or *case-construct*
or *do-construct*
or *enable-construct*
or *if-construct*
or *where-construct*

R219	*action-stmt*	**is**	*allocate-stmt*
		or	*assignment-stmt*
		or	*backspace-stmt*
		or	*call-stmt*
		or	*close-stmt*
		or	*continue-stmt*
		or	*cycle-stmt*
		or	*deallocate-stmt*
		or	*endfile-stmt*
		or	*exit-stmt*
		or	*forall-stmt*
		or	*goto-stmt*
		or	*identify-stmt*
		or	*if-stmt*
		or	*inquire-stmt*
		or	*open-stmt*
		or	*print-stmt*
		or	*read-stmt*
		or	*return-stmt*
		or	*rewind-stmt*
		or	*set-range-stmt*
		or	*signal-stmt*
		or	*stop-stmt*
		or	*where-stmt*
		or	*write-stmt*
		or	*arithmetic-if-stmt*
		or	*assign-stmt*
		or	*assigned-goto-stmt*
		or	*computed-goto-stmt*
		or	*pause-stmt*

Constraint: An *entry-stmt* or *return-stmt* must not appear in a main program; an *entry-stmt* must not appear in constructs.

3 LEXICAL ELEMENTS

R301	*character*	**is**	*alphanumeric-character*
		or	*special-character*
R302	*alphanumeric-character*	**is**	*letter*
		or	*digit*
		or	*underscore*
R303	*symbolic-name*	**is**	*letter* [*alphanumeric-character*]...

Constraint: The maximum length of a *symbolic-name* is 31 characters.

R304	*constant*	**is**	*literal-constant*
		or	*symbolic-constant*
R305	*literal-constant*	**is**	*int-constant*
		or	*real-constant*

		or	*complex-constant*
		or	*logical-constant*
		or	*char-constant*
		or	*bit-constant*
R306	*symbolic-constant*	**is**	*symbolic-name*
R307	*intrinsic-operator*	**is**	*power-op*
		or	*mult-op*
		or	*add-op*
		or	*bnot-op*
		or	*band-op*
		or	*bor-op*
		or	*concat-op*
		or	*rel-op*
		or	*not-op*
		or	*and-op*
		or	*or-op*
		or	*equiv-op*
R308	*power-op*	**is**	**
R309	*mult-op*	**is**	*
		or	/
R310	*add-op*	**is**	+
		or	−
R311	*bnot-op*	**is**	.BNOT.
R312	*band-op*	**is**	.BAND.
R313	*bor-op*	**is**	.BOR.
		or	.BXOR.
R314	*concat-op*	**is**	//
R315	*rel-op*	**is**	.EQ.
		or	.NE.
		or	.LT.
		or	.LE.
		or	.GT.
		or	.GE.
		or	==
		or	<>
		or	<
		or	<=
		or	>
		or	>=
R316	*not-op*	**is**	.NOT.
R317	*and-op*	**is**	.AND.
R318	*or-op*	**is**	.OR.
R319	*equiv-op*	**is**	.EQV.
		or	.NEQV.
R320	*defined-operator*	**is**	*overloaded-intrinsic-op*

		or *defined-unary-op*
		or *defined-binary-op*
R321	*overloaded-intrinsic-op*	**is** *intrinsic-operator*
R322	*defined-unary-op*	**is** . *letter* [*letter*]... .
R323	*defined-binary-op*	**is** . *letter* [*letter*]... .

Constraint: A *defined-unary-op* and a *defined-binary-op* must not contain more than 31 characters and must not be the same as any *intrinsic-operator* or *logical-constant*.

R324	*label*	**is** *digit* [*digit* [*digit* [*digit* [*digit*]]]]

4 DATA TYPES

R401	*int-constant*	**is** *digit* [[_] *digit*]...
R402	*signed-int-constant*	**is** [*sign*] *int-constant*
R403	*sign*	**is** +
		or −
R404	*signed-real-constant*	**is** [*sign*] *real-constant*
R405	*real-constant*	**is** *significand* [*exponent-letter exponent*]
		or *int-constant exponent-letter exponent*
R406	*significand*	**is** *int-constant* . [*int-constant*]
		or . *int-constant*
R407	*exponent*	**is** *signed-int-constant*
R408	*exponent-letter*	**is** E
		or D
		or *defined-exponent-letter*
R409	*exponent-letter-stmt*	**is** EXPONENT LETTER *precision-selector defined-exponent-letter*
R410	*defined-exponent-letter*	**is** *letter*

Constraint: A *defined-exponent-letter* must be a letter other than E, D, or H.

R411	*complex-constant*	**is** (*real-part* , *imag-part*)
R412	*real-part*	**is** *signed-int-constant*
		or *signed-real-constant*
R413	*imag-part*	**is** *signed-int-constant*
		or *signed-real-constant*
R414	*char-constant*	**is** ' [*character*]... '
		or " [*character*]... "
R415	*logical-constant*	**is** .TRUE.
		or .FALSE.
R416	*bit-constant*	**is** B"0"
		or B'0'
		or B"1"
		or B'1'

R417 *derived-type-def* **is** *derived-type-stmt*
component-def-stmt
[*component-def-stmt*]...
[*variant-component*]
end-type-stmt

R418 *derived-type-stmt* **is** [*access-spec*] TYPE *type-name* [(*type-param-name-list*)]

R419 *end-type-stmt* **is** END TYPE [*type-name*]

Constraint: A derived type *type-name* must not be the same as any intrinsic *type-name*.

Constraint:

R420 *component-def-stmt* **is** *type-spec* [[, *component-attr-spec*]... ::] *component-decl-list*

Constraint: A *type-spec* in a *component-def-stmt* must not contain a *type-param-value* that is an asterisk.

R421 *component-attr-spec* **is** PRIVATE
or ARRAY (*explicit-shape-spec-list*)

R422 *component-decl* **is** *component-name* [(*explicit-shape-spec-list*)]

R423 *variant-component* **is** *select-case-stmt*
[*case-stmt* [*component-def-stmt*]...]...
end-select-stmt

Constraint: The *select-case-stmt* and *end-select-stmt* in a *variant-component* must not specify a *construct-name*.

R424 *derived-type-constructor* **is** *type-name* [(*type-param-spec-list*)] (*expr-list*)

Constraint: The *type-param-spec* option must be supplied if and only if the referenced type definition includes type parameters.

R425 *array-constructor* **is** [*array-constructor-value-list*]
or (/ *constructor-value-list* /)

R426 *array-constructor-value* **is** *scalar-expr*
or *rank-1-array-expr*
or *scalar-int-expr* : *scalar-int-expr* [: *scalar-int-expr*]
or [*int-constant-expr*] *array-constructor*

5 DATA OBJECT DECLARATIONS AND SPECIFICATIONS

R501 *type-declaration-stmt* **is** *type-spec* [[, *attr-spec*]... ::] *object-decl-list*

R502 *type-spec* **is** INTEGER
or REAL [*precision-selector*]
or DOUBLE PRECISION
or COMPLEX [*precision-selector*]
or CHARACTER [*length-selector*]
or LOGICAL
or BIT
or TYPE (*type-name* [(*type-param-spec-list*)])

R503 *type-param-spec* **is** [*type-param-name* =] *type-param-value*

R504 *type-param-value* **is** *specification-expr*

		or *
R505	*attr-spec*	**is** *value-spec*
		or *access-spec*
		or ALIAS
		or ALLOCATABLE
		or ARRAY (*array-spec*)
		or INTENT (*intent-spec*)
		or OPTIONAL
		or RANGE
		or SAVE
R506	*object-decl*	**is** *object-name* [(*array-spec*)] [* *char-length*] [= *constant-expr*]

Constraint: No *attr-spec* may appear more than once in a given *type-declaration-stmt*.

Constraint:

Constraint: The = *constant-expr* must appear if and only if the statement contains a *value-spec* attribute (5.1.2.1, 7.1.6.1).

Constraint: The * *char-length* option is permitted only if the *type-spec* is CHARACTER. If present **char-length* overrides the *length-selector* for that specific *object-decl* in which it appears.

Constraint: The ALLOCATABLE and RANGE attributes may be used only when declaring array objects.

Constraint:

Constraint: An array specified with an ALIAS attribute must be declared with an *allocatable-spec*.

Constraint:

R507	*precision-selector*	**is** (*type-param-value* [, [EXPONENT_RANGE =] □
		□ *type-param-value*]
		or (PRECISION = *type-param-value* □
		□ [, EXPONENT_RANGE = *type-param-value*])
		or (EXPONENT_RANGE = *type-param-value* □
		□ [, PRECISION = *type-param-value*])

Constraint: The *type-param-value* must be an integer constant expression or an asterisk.

R508	*length-selector*	**is** [LEN =] *type-param-value*
		or * *char-length [,]*
R509	*char-length*	is (*type-param-value*)
		or *scalar-int-constant*
R510	*value-spec*	**is** PARAMETER
		or INITIAL
R511	*access-spec*	**is** PUBLIC
		or PRIVATE
R512	*intent-spec*	**is** IN
		or OUT
		or INOUT
R513	*array-spec*	**is** *explicit-shape-spec-list*
		or *assumed-shape-spec-list*

or *allocatable-spec-list*
or *assumed-size-spec*

R514 *explicit-shape-spec* **is** [*lower-bound* :] *upper-bound*

R515 *lower-bound* **is** *specification-expr*

R516 *upper-bound* **is** *specification-expr*

Constraint: An explicit shape array whose bounds depend on the values of variables must either be a dummy argument or a local array of a procedure.

R517 *assumed-shape-spec* **is** [*lower-bound*] :

R518 *allocatable-spec* **is** :

R519 *assumed-size-spec* is [*explicit-shape-spec-list* ,] [*lower-bound* :] *

Constraint: *assumed-size-spec* must not be included in an ARRAY attribute.

Constraint:

R520 *intent-stmt* **is** INTENT (*intent-spec*) [::] *dummy-arg-name-list*

R521 *optional-stmt* **is** OPTIONAL [::] *dummy-arg-name-list*

R522 *access-stmt* **is** *access-spec* [[::] *object-name-list*]

Constraint: An *access-stmt* may appear only in a module and only one accessibility statement with omitted object name list is permitted in a host program unit.

R523 *save-stmt* **is** SAVE [[::] *saved-object-list*]

R524 *saved-object* **is** *object-name*
or / *common-block-name* /

Constraint: An object name must not be a dummy argument name, a procedure name, a function result name, an automatic array name, an alias name,

Constraint: If a SAVE statement with an omitted saved object list occurs in a program unit, no other occurrence of the SAVE attribute or SAVE statement is permitted.

R525 *dimension-stmt* is DIMENSION *array-name* (*array-spec*) [, *array-name* (*array-spec*)]...

Constraint: In a DIMENSION statement, only explicit shape and assumed-size *array-specs* are permitted.

R526 *initialize-stmt* **is** INITIALIZE (*initial-value-def-list*)

R527 *initial-value-def* **is** *variable* = *constant-expr*

R528 *parameter-stmt* **is** PARAMETER (*symbolic-constant-def-list*)

R529 *symbolic-constant-def* **is** *symbolic-constant-name* = *constant-expr*

R530 *condition-stmt* **is** CONDITION [::] *condition-name-list*

R531 *range-stmt* **is** RANGE [/ *range-list-name* /] *array-name-list*

R532 *implicit-stmt* **is** IMPLICIT *implicit-spec-list*
or IMPLICIT NONE

R533 *implicit-spec* **is** *type-spec* (*letter-spec-list*)

R534 *letter-spec* **is** *letter* [– *letter*]

R535 *equivalence-stmt* is EQUIVALENCE *equivalence-set-list*

R536 *equivalence-set* is (*equivalence-object* , *equivalence-object-list*)

R537	*equivalence-object*	is	*object-name*
		or	*array-element*
		or	*substring*

Constraint: *object-name* must be a scalar variable name or an array variable name.

Constraint:

Constraint: Each subscript or substring range expression in an *equivalence-object* must be an integer constant expression.

R538	*common-stmt*	is	COMMON [/ [*common-block-name*] /]*common-block-object-list* □
			□ [[,] / [*common-block-name*] /*common-block-object-list*]...
R539	*common-block-object*	is	*object-name* [(*explicit-shape-spec-list*)]

Constraint: *object-name* must be a *scalar-variable-name* or an *array-variable-name*. Only one appearance of object name is permitted in all common block object lists within a program unit.

Constraint:

Constraint: Each bound in the *explicit-shape-spec* must be an integer constant expression.

R540	*data-stmt*	is	DATA *data-stmt-init* [[,] *data-stmt-init*]...
R541	*data-stmt-init*	is	*data-stmt-object-list* / *data-stmt-value-list* /
R542	*data-stmt-object*	is	*object-name*
		or	*array-element*
		or	*data-implied-do*
R543	*data-stmt-value*	is	[*data-stmt-repeat* *] *data-stmt-constant*
R544	*data-stmt-constant*	is	*constant*
		or	*signed-int-constant*
		or	*signed-real-constant*
R545	*data-stmt-repeat*	is	*int-constant*
		or	*scalar-int-symbolic-constant*
R546	*data-implied-do*	is	(*data-i-do-object-list*, *do-variable* = *scalar-int-expr*, *scalar-int-expr* [, *scalar-int-expr*])
R547	*data-i-do-object*	is	*array-element*
		or	*data-implied-do*

6 USE OF DATA OBJECTS

R601	***variable***	**is**	***scalar-variable-name***
		or	***array-variable-name***
		or	***array-element***
		or	***array-section***
		or	***structure-component***
		or	***substring***
R602	***substring***	**is**	***parent-string* (*substring-range*)**
R603	***parent-string***	**is**	***char-scalar-variable-name***
		or	***char-array-element***

		or *scalar-char-structure-component*
		or *scalar-char-symbolic-constant*
		or *scalar-char-constant*
R604	*substring-range*	**is** [*scalar-int-expr*] : [*scalar-int-expr*]
R605	*structure-component*	**is** *parent-structure* % *component-name* [*array-selector*]
R606	*parent-structure*	**is** *derived-type-scalar-variable-name*
		or *derived-type-array-variable-name*
		or *derived-type-array-element*
		or *derived-type-array-section*
		or *derived-type-structure-component*
		or *derived-type-symbolic-constant*

Constraint: An *array-selector* may appear only if the component specified by *component-name* is an array.

R607	*array-selector*	**is** (*subscript-list*)
		or (*section-subscript-list*)
R608	*allocate-stmt*	**is** ALLOCATE (*array-allocation-list*)
R609	*array-allocation*	**is** *array-name* (*explicit-shape-spec-list*)

Constraint: *array-name* must be the name of an allocatable array.

Constraint:

Constraint: The number of *explicit-shape-spec*s in an *array-allocation explicit-shape-spec-list* must be the same as the declared rank of the array.

R610	*deallocate-stmt*	**is** DEALLOCATE (*array-name-list*)
R611	*array-element*	**is** *parent-array* (*subscript-list*)

Constraint: The number of subscripts must equal the declared rank of the array.

R612	*array-section*	**is** *parent-array* (*section-subscript-list*) [(*substring-range*)]
R613	*parent-array*	**is** *array-variable-name*
		or *array-symbolic-constant-name*

Constraint: At least one *section-subscript* must be a *subscript-triplet* or a *vector-int-expr*.

Constraint:

R614	*subscript*	**is** *scalar-int-expr*
R615	*section-subscript*	**is** *subscript*
		or *subscript-triplet*
		or *vector-int-expr*

Constraint: A *vector-int-expr section-subscript* must be a rank one integer array.

R616	*subscript-triplet*	**is** [*subscript*] : [*subscript*] [: *stride*]
R617	*stride*	**is** *scalar-int-expr*
R618	*set-range-stmt*	**is** SET RANGE ([*effective-range-list*]) *array-name-list*
		or SET RANGE ([*effective-range-list*]) / *range-list-name* /
R619	*effective-range*	**is** *explicit-shape-spec*
		or [*lower-bound*] : [*upper-bound*]

Constraint: The number of effective ranges in an *effective-range-list* must equal the rank of the arrays being ranged.

Constraint:

Constraint: An array that is a member of a range list must not appear in an *array-name-list* of a SET RANGE statement.

R620 *identify-stmt* **is** IDENTIFY (*alias-name* = *parent*)
or IDENTIFY (*alias-element* = *parent-element* , □
□ *alias-range-spec-list*)

Constraint: The alias and parent objects must conform in type, rank, and type parameters.

Constraint:

R621 *alias-element* **is** *alias-name* (*subscript-range*)

Constraint: The number of *subscript-name*s in an alias element must equal the number of *alias-range-spec*s.

Constraint:

R622 *parent-element* **is** *parent-name* (*subscript-mapping*) [% *component-name* [(*subscript-list*)]]...

R623 *subscript-mapping* **is** *subscript-list*

Constraint: Each *subscript* must be linear in the *alias-element subscript-name*s.

R624 *alias-range-spec* **is** *subscript-range* = *subscript* : *subscript*

Constraint: The subscript ranges in a *subscript-name-list* must be identical to the subscript ranges in the corresponding alias range specification list, and must appear in the same order. A name must not appear more than once in such a list.

Constraint:

7 EXPRESSIONS AND ASSIGNMENT

R701 *primary* **is** *constant*
or *variable*
or *array-constructor*
or *derived-type-constructor*
or *function-reference*
or (*expr*)

R702 *level-1-expr* **is** [*defined-unary-op*] *primary*

R322 *defined-unary-op* **is** . *letter* [*letter*]... .

R703 *mult-operand* **is** *level-1-expr* [*power-op mult-operand*]

R704 *add-operand* **is** [*add-operand mult-op*] *mult-operand*

R705 *level-2-expr* **is** [*add-op*] *add-operand*
or *level-2-expr add-op add-operand*

R308 *power-op* **is** **

R309 *mult-op* **is** *
or /

R310	*add-op*	**is**	+
		or	–
R706	*band-operand*	**is**	[*bnot-op*] *level-2-expr*
R707	*bor-operand*	**is**	[*bor-operand band-op*] *band-operand*
R708	*level-3-expr*	**is**	[*level-3-expr bor-op*] *bor-operand*
R311	*bnot-op*	**is**	.BNOT.
R312	*band-op*	**is**	.BAND.
R313	*bor-op*	**is**	.BOR.
		or	.BXOR.
R709	*level-4-expr*	**is**	[*level-4-expr concat-op*] *level-3-expr*
R314	*concat-op*	**is**	//
R710	*level-5-expr*	**is**	[*level-4-expr rel-op*] *level-4-expr*
R315	*rel-op*	**is**	.EQ.
		or	.NE.
		or	.LT.
		or	.LE.
		or	.GT.
		or	.GE.
		or	==
		or	<>
		or	<
		or	<=
		or	>
		or	>=
R711	*and-operand*	**is**	[*not-op*] *level-5-expr*
R712	*or-operand*	**is**	[*or-operand and-op*] *and-operand*
R713	*equiv-operand*	**is**	[*equiv-operand or-op*] *or-operand*
R714	*level-6-expr*	**is**	[*level-6-expr equiv-op*] *equiv-operand*
R316	*not-op*	**is**	.NOT.
R317	*and-op*	**is**	.AND.
R318	*or-op*	**is**	.OR.
R319	*equiv-op*	**is**	.EQV.
		or	.NEQV.
R715	*expr*	**is**	[*expr defined-binary-op*] *level-6-expr*
R323	*defined-binary-op*	**is**	. *letter* [*letter*]... .
R716	*assignment-stmt*	**is**	*variable* = *expr*
R717	*where-stmt*	**is**	WHERE (*array-mask-expr*) *array-assignment-stmt*
R718	*where-construct*	**is**	*where-construct-stmt*
			[*array-assignment-stmt*]...
			[*elsewhere-stmt*
			[*array-assignment-stmt*]...]
			end-where-stmt

R719 *where-construct-stmt* **is** WHERE (*array-mask-expr*)

R720 *array-mask-expr* **is** *logical-expr*
or *bit-expr*

R721 *elsewhere-stmt* **is** ELSEWHERE

R722 *end-where-stmt* **is** END WHERE

Constraint: The shape of the *array-mask-expr* and the variable being defined in each *array-assignment-stmt* must be the same.

R723 *forall-stmt* **is** FORALL (*forall-triplet-spec-list* [,*scalar-mask-expr*]) *forall-assignment*

R724 *forall-triplet-spec* **is** *subscript-name* = *subscript* : *subscript* [: *stride*]

Constraint: *subscript-name* must be a *scalar-symbolic-name* of type integer.

Constraint:

R725 *forall-assignment* **is** *array-element* = *expr*
or *array-section* = *expr*

Constraint: The *array-section* or *array-element* must contain references to all subscript names in the *forall-triplet-spec-list*. *expr* in a *forall-assignment* must reference all of the *forall-triplet-spec subscript-names*.

8 EXECUTION CONTROL

R801 *block* **is** [*execution-part*]...

R802 *if-construct* **is** *if-then-stmt*
block
[*else-if-stmt*
block]...
[*else-stmt*
block]
end-if-stmt

R803 *if-then-stmt* **is** [*if-construct-name* :] IF (*scalar-mask-expr*) THEN

R804 *else-if-stmt* **is** ELSE IF (*scalar-mask-expr*) THEN

R805 *else-stmt* **is** ELSE

R806 *end-if-stmt* **is** END IF [*if-construct-name*]

Constraint: If an *if-construct-name* is present, the same name must be specified on both the *if-then-stmt* and the corresponding *end-if-stmt*.

R807 *if-stmt* **is** IF (*scalar-mask-expr*) *action-stmt*

Constraint: The *action-stmt* in the *if-stmt* must not be an *if-stmt*.

R808 *case-construct* **is** *select-case-stmt*
[*case-stmt*
block]...
end-select-stmt

R809 *select-case-stmt* **is** [*select-construct-name* :] SELECT CASE (*case-expr*)

R810 *case-stmt* **is** CASE *case-selector*

R811 *end-select-stmt* **is** END SELECT [*select-construct-name*]

Constraint: If a *select-construct-name* is present, the same name must be specified on both the *select-case-stmt* and the corresponding *end-select-stmt*.

R812 *case-expr* **is** *scalar-int-expr*
or *scalar-char-expr*
or *scalar-logical-expr*
or *scalar-bit-expr*

R813 *case-selector* **is** (*case-value-range-list*)
or DEFAULT

Constraint: Only one DEFAULT *case-selector* may appear in any given *case-construct*.

R814 *case-value-range* **is** *case-value*
or [*case-value*] : [*case-value*]

R815 *case-value* **is** *scalar-int-constant-expr*
or *scalar-char-constant-expr*
or *scalar-logical-constant-expr*
or *scalar-bit-constant-expr*

R816 *do-construct* **is** *do-stmt*
do-body
do-termination

R817 *do-stmt* **is** [*do-construct-name* :] DO [*label*] [[,] *loop-control*]

R818 *loop-control* **is** *do-variable* = *scalar-numeric-expr, scalar-numeric-expr* [, *scalar-numeric-expr*]
or (*scalar-int-expr* TIMES)

Constraint: The *do-variable* must be a scalar integer, real, or double precision variable.

R819 *do-body* **is** [*execution-part*]...

R820 *do-termination* **is** *end-do-stmt*
or *continue-stmt*
or *do-term-stmt*
or *do-construct*

R821 *do-term-stmt* **is** *action-stmt*

Constraint: If the *label* is omitted in a *do-stmt*, the corresponding *do-termination* must be an *end-do-stmt*.

Constraint:

Constraint: If the *do-termination* is a *continue-stmt*

Constraint: A *do-term-stmt* must not be a *continue-stmt*, *goto-stmt*, *return-stmt*, *stop-stmt*, *exit-stmt*, *cycle-stmt*, *arithmetic-if-stmt*, *assigned-goto-stmt*, *computed-goto-stmt*, nor an *if-stmt* that causes a transfer of control.

Constraint:

Constraint: If a *do-termination* is a *do-construct*, the *do-termination* of that *do-construct* must not be an *end-do-stmt*.

R822 *end-do-stmt* **is** END DO [*do-construct-name*]

Constraint: If a *do-construct-name* is used on the *do-stmt*, the corresponding *do-termination* must be an *end-do-stmt* that uses the same *do-construct-name*. If a *do-construct-name* does not appear on the *do-stmt*, a *do-construct-name* must not appear on the corresponding *do-termination*.

R823 *exit-stmt* **is** EXIT [*do-construct-name*]

R824 *cycle-stmt* **is** CYCLE [*do-construct-name*]

R825 *enable-construct* **is** *enable-stmt*
        block
      [*handle-stmt*
        block]...
      end-enable-stmt

R826 *enable-stmt* **is** [*enable-construct-name* :] ENABLE [(*condition-name-list*)]

R827 *handle-stmt* **is** HANDLE (*condition-name-list*)
  or HANDLE (*)

R828 *end-enable-stmt* **is** END ENABLE [*enable-construct-name*]

Constraint: A *condition-name* must not appear more than once in a single *condition-name-list*.

Constraint:

Constraint: HANDLE (*) may appear at most once in an ENABLE construct.

Constraint:

R829 *signal-stmt* **is** SIGNAL (*condition-name*)
  or SIGNAL (*)

Constraint: SIGNAL (*) is permitted only in a HANDLE block.

R830 *goto-stmt* **is** GO TO *label*

Constraint: *label* must be the statement label of a *branch-target* that appears in the same program unit as the *go-to-stmt*.

R831 *computed-goto-stmt* is GO TO (*label-list*) [,] *scalar-int-expr*

Constraint: Each *label* in *label-list* must be the statement label of a branch target that appears in the same program unit as the *computed-goto-stmt*.

R832 *assign-stmt* is ASSIGN *label* TO *scalar-int-variable*

Constraint: *label* must be the statement label of a branch target or a *format-stmt*.

R833 *assigned-goto-stmt* is GO TO *scalar-int-variable* [[,] (*label-list*)]

Constraint: Each *label* in *label-list* must be the statement label of a branch target that appears in the same program unit as the *assigned-goto-stmt*.

R834 *arithmetic-if-stmt* is IF (*scalar-numeric-expr*) *label*, *label*, *label*

Constraint: Each *label* must be the label of a branch target that appears in the same program unit as the *arithmetic-if-stmt*.

Constraint:

R835 *continue-stmt* **is** CONTINUE

R836 *stop-stmt* **is** STOP [*access-code*]

R837 *access-code* **is** *char-constant*

		or *digit* [*digit* [*digit* [*digit* [*digit*]]]]
R838	*pause-stmt*	**is** PAUSE [*access-code*]

9 INPUT/OUTPUT STATEMENTS

R901	*io-unit*	**is** *external-file-unit*
		or *
		or *internal-file-unit*
R902	*external-file-unit*	**is** *scalar-int-expr*
R903	*internal-file-unit*	**is** *char-variable*
R904	*open-stmt*	**is** OPEN (*connect-spec-list*)
R905	*connect-spec*	**is** [UNIT=] *external-file-unit*
		or IOSTAT= *iostat-variable*
		or ERR= *label*
		or FILE= *scalar-char-expr*
		or STATUS= *scalar-char-expr*
		or ACCESS= *scalar-char-expr*
		or FORM= *scalar-char-expr*
		or RECL= *scalar-int-expr*
		or BLANK= *scalar-char-expr*
		or POSITION= *scalar-char-expr*
		or ACTION= *scalar-char-expr*
		or DELIM= *scalar-char-expr*
		or PAD= *scalar-char-expr*

Constraint: Each specifier must not appear more than once in a given *open-stmt*; the UNIT= specifier must appear.

Constraint:

Constraint: If the STATUS= specifier is 'SCRATCH', the FILE= specifier must be absent.

Constraint:

R906	*close-stmt*	**is** CLOSE (*close-spec-list*)
R907	*close-spec*	**is** [UNIT=] *external-file-unit*
		or IOSTAT= *iostat-variable*
		or ERR= *label*
		or STATUS= *scalar-char-expr*

Constraint: A given specifier must not appear more than once in a given *close-stmt*; the unit specifier must appear.

Constraint:

R908	*read-stmt*	**is** READ (*io-control-spec-list*) [*input-item-list*]
		or READ *format* [, *input-item-list*]
R909	*write-stmt*	**is** WRITE (*io-control-spec-list*) [*output-item-list*]
R910	*print-stmt*	**is** PRINT *format* [, *output-item-list*]
R911	*io-control-spec*	**is** [UNIT=] *io-unit*

or [FMT=] *format*
or REC= *scalar-int-expr*
or PROMPT= *scalar-char-expr*
or IOSTAT= *iostat-variable*
or ERR= *label*
or END= *label*
or NULLS= *scalar-int-variable*
or VALUES= *scalar-int-variable*

Constraint: An *io-control-spec-list* must contain exactly one *io-unit* and may contain at most one of each of the other specifiers.

Constraint:

R912 *format* **is** *char-expr*
or *label*
or *
or **
or *scalar-int-variable*

R913 *iostat-variable* **is** *scalar-int-variable*

R914 *input-item* **is** *variable*
or *io-implied-do*

R915 *output-item* **is** *expr*
or *io-implied-do*

R916 *io-implied-do* **is** (*io-implied-do-object-list* , *io-implied-do-control*)

R917 *io-implied-do-object* **is** *input-item*
or *output-item*

R918 *io-implied-do-control* **is** *scalar-numeric-expr* , ☐
☐ *scalar-numeric-expr* , [*scalar-numeric-expr*]

Constraint: In an *input-item-list*, an *io-implied-do-object* must be an *input-item*. In an *output-item-list*, an *io-implied-do-object* must be an *output-item*.

Constraint:

Constraint: The *do-variable* of an *io-implied-do* that is contained within another *io-implied-do* must not appear as, nor be associated with, the *do-variable* of the containing *io-implied-do*.

R919 *backspace-stmt* **is** BACKSPACE *external-file-unit*
or BACKSPACE (*position-spec-list*)

R920 *endfile-stmt* **is** ENDFILE *external-file-unit*
or ENDFILE (*position-spec-list*)

R921 *rewind-stmt* **is** REWIND *external-file-unit*
or REWIND (*position-spec-list*)

Constraint: BACKSPACE, ENDFILE, and REWIND apply only to files connected for sequential access.

R922 *position-spec* **is** [UNIT =] *external-file-unit*
or IOSTAT = *iostat-variable*
or ERR = *label*

Constraint: A *position-spec-list* must contain exactly one *external-file-unit* and may contain at most one of each of the other specifiers.

R923 *inquire-stmt* **is** INQUIRE (*inquire-spec-list*) [*output-item-list*]

R924 *inquire-spec* **is** FILE = *scalar-char-expr*
or UNIT = *external-file-unit*
or IOSTAT = *iostat-variable*
or ERR = *label*
or EXIST = *scalar-logical-variable*
or OPENED = *scalar-logical-variable*
or NUMBER = *scalar-int-variable*
or NAMED = *scalar-logical-variable*
or NAME = *scalar-char-variable*
or ACCESS = *scalar-char-variable*
or SEQUENTIAL = *scalar-char-variable*
or DIRECT = *scalar-char-variable*
or FORM = *scalar-char-variable*
or FORMATTED = *scalar-char-variable*
or UNFORMATTED = *scalar-char-variable*
or RECL = *scalar-int-variable*
or NEXTREC = *scalar-int-variable*
or BLANK = *scalar-char-variable*
or POSITION = *scalar-char-variable*
or ACTION = *scalar-char-variable*
or DELIM = *scalar-char-variable*
or PAD = *scalar-char-variable*
or IOLENGTH = *scalar-int-variable*

Constraint: An INQUIRE statement must contain one FILE= specifier or one UNIT= specifier, but not both, and at most one of each of the other specifiers.

Constraint:

10 INPUT/OUTPUT EDITING

R1001 *format-stmt* **is** FORMAT *format-specification*

R1002 *format-specification* **is** ([*format-item-list*])

Constraint: The *format-stmt* must be labeled.

Constraint:

R1003 *format-item* **is** [*r*] *data-edit-desc*
or *control-edit-desc*
or *char-string-edit-desc*
or [*r*] (*format-item-list*)

R1004 *r* **is** *int-constant*

Constraint: *r* must be positive. It is called a

Constraint: The character underscore (_) is prohibited in an *int-constant* in a format specification.

R1005 *data-edit-desc*	**is** I *w* [. *m*]
	or F *w* . *d*
	or E *w* . *d* [E *e*]
	or EN *w* . *d* [E *e*]
	or G *w* . *d* [E *e*]
	or B *w*
	or L *w*
	or A [*w*]
	or D *w* . *d*
R1006 *w*	**is** *scalar-int-constant*
R1007 *m*	**is** *scalar-int-constant*
R1008 *d*	**is** *scalar-int-constant*
R1009 *e*	**is** *scalar-int-constant*

Constraint: *w* and *e* must be positive and *d* and *m* must be zero or positive.

Constraint:

R1010 *control-edit-desc*	**is** *position-edit-desc*
	or [*r*] /
	or :
	or *sign-edit-desc*
	or *k* P
	or *blank-interp-edit-desc*
R1011 *k*	**is** *scalar-signed-int-constant*
R1012 *position-edit-desc*	**is** T *n*
	or TL *n*
	or TR *n*
	or *n* X
R1013 *n*	**is** *scalar-int-constant*
R1014 *sign-edit-desc*	**is** S
	or SP
	or SS
R1015 *blank-interp-edit-desc*	**is** BN
	or BZ
R1016 *char-string-edit-desc*	**is** *char-constant*
	or *c* H *character* [*character*]...
R1017 *c*	**is** *int-constant*

Constraint: *c* must be positive.

11 PROGRAM UNITS

R200 *main-program*	**is** [*program-stmt*] *program-unit-body* *end-program-stmt*

R1101	*program-stmt*	**is** PROGRAM *program-name*
R1102	*end-program-stmt*	**is** END [PROGRAM [*program-name*]]

Constraint: The *program-name* may be included in the *end-program-stmt* only if the optional *program-stmt* is used and, if included, must be identical to the *program-name* specified in the *program-stmt*.

R200	*module-subprogram*	**is** *module-stmt* *program-unit-body* *end-module-stmt*
R1103	*module-stmt*	**is** MODULE *module-name*
R1104	*end-module-stmt*	**is** END [MODULE [*module-name*]]

Constraint: If the *module-name* is specified in the *end-module-stmt*, it must be identical to the *module-name* specified in the *module-stmt*.

R1105	*use-stmt*	**is** USE [*module-name*] [[,] *all-clause*]
		or USE [*module-name*] [,] ONLY ([*only-list*])
R1106	*all-clause*	**is** ALL ([*rename-list*])
		or ALL [([*rename-list*])] EXCEPT (*except-list*)
R1107	*rename*	**is** *use-name* => *local-name*
R1108	*except*	**is** *use-name*
R1109	*only*	**is** *use-name* [=> *local-name*]
R1110	*use-name*	**is** *variable-name*
		or *procedure-name*
		or *type-name*
		or *condition-name*
		or *constant-name*
R1111	*block-data-subprogram*	is *block-data-stmt*
R1112	*block-data-stmt*	is BLOCK DATA [*block-data-name*]
R1113	*end-block-data-stmt*	is END [BLOCK DATA [*block-data-name*]]

Constraint: The *block-data-name* may be included in the *end-block-data-stmt* only if it was provided in the *block-data-stmt* and, if included, must be identical to the *block-data-name* in the *block-data-stmt*.

12 PROCEDURES

R1201	*interface-block*	**is** *interface-stmt* *interface-specification* *end-interface-stmt*
R1202	*interface-stmt*	**is** INTERFACE
R1203	*end-interface-stmt*	**is** END INTERFACE
R1204	*interface-specification*	**is** *interface-header* [*use-stmt*]... [*implicit-part*]... [*declaration-part*]...

R1205	*interface-header*	**is** *function-stmt* **or** *subroutine-stmt*
R1206	*external-stmt*	**is** EXTERNAL *external-name-list*
R1207	*external-name*	**is** *external-procedure-name* **or** *dummy-arg-name* or *block-data-name*
R1208	*intrinsic-stmt*	**is** INTRINSIC *intrinsic-name-list*
R1209	*intrinsic-name*	**is** *intrinsic-procedure-name* **or** *intrinsic-condition-name*
R1210	*function-reference*	**is** *function-name* ([*actual-arg-spec-list*])
Constraint:		The *actual-arg-spec-list* for a function reference must not contain an *alt-return-spec*.
R1211	*call-stmt*	**is** CALL *subroutine-name* [([*actual-arg-spec-list*])]
R1212	*actual-arg-spec*	**is** [*keyword* =] *actual-arg*
R1213	*keyword*	**is** *dummy-arg-name*
R1214	*actual-arg*	**is** *expr* **or** *variable* **or** *procedure-name* **or** *condition-name* or *alt-return-spec*
R1215	*alt-return-spec*	is * *label*
Constraint:		The *keyword* may be omitted from an *actual-arg-spec* only if the *keyword* has been omitted from each preceding *actual-arg-spec* in the argument list.
Constraint:		
R207	*function-subprogram*	**is** *function-stmt* *program-unit-body* *end-function-stmt*
R1216	*function-stmt*	**is** [*prefix*] FUNCTION *function-name* ([*dummy-arg-name-list*]) [*suffix*]
R1217	*prefix*	**is** *type-spec* [RECURSIVE] **or** RECURSIVE [*type-spec*]
R1218	*suffix*	**is** RESULT (*result-name*) [OPERATOR (*defined-operator*)] **or** OPERATOR (*defined-operator*) [RESULT (*result-name*)]
R1219	*end-function-stmt*	**is** END [FUNCTION [*function-name*]]
Constraint:		FUNCTION must be present on the *end-function-stmt* of an internal function.
Constraint:		
R1220	*subroutine-subprogram*	**is** *subroutine-stmt* *program-unit-body* *end-subroutine-stmt*
R1221	*subroutine-stmt*	**is** [RECURSIVE] SUBROUTINE *subroutine-name* □ □ [(*dummy-arg-list*)] [ASSIGNMENT]
R1222	*dummy-arg*	**is** *dummy-arg-name* or *

R1223 *end-subroutine-stmt* **is** END [SUBROUTINE [*subroutine-name*]]

Constraint: SUBROUTINE must be present on the end of an internal subroutine.

Constraint:

R1224 *entry-stmt* is ENTRY *entry-name* [([*dummy-arg-list*])]

Constraint: A *dummy-arg* may be an alternate return indicator only if the ENTRY statement is contained in a subroutine subprogram.

R1225 *return-stmt* **is** RETURN [*scalar-int-expr*]

Constraint: The *return-stmt* must be contained in a function or subroutine subprogram.

Constraint:

Constraint: The *expression* must produce a scalar result of type integer.

R1226 *stmt-function-stmt* is *function-name* ([*dummy-arg-name-list*]) = *expr*

Constraint: The *expr* may be composed only of constants (literal and symbolic), references to scalar variables and array elements, references to functions, and intrinsic operators. If a reference to another statement function appears in *expr*, its definition must have been provided earlier in the program unit.

Constraint:

13 INTRINSIC PROCEDURES

14 ENTITY SCOPE, ASSOCIATION, AND DEFINITION

15 DEPRECATED FEATURES

13 INTRINSIC PROCEDURES

APPENDIX E PERMUTED INDEX FOR HEADINGS

1.6 Deprecated Features, Core	Conformance
7.5.1.4 Intrinsic Assignment	Conformance Rules
9.3 File	Connection
9.3.2	Connection of a File to a Unit
2.4.3.3	Constant
10.7.1 Character	Constant Edit Descriptor
7.1.6.1	Constant Expression
3.2.3	Constants
4.1.2	Constants
6.2.1.1 Array	Constants and Variables
8.1.2 IF	Construct
8.1.2.1 Form of the IF	Construct
8.1.2.2 Execution of an IF	Construct
8.1.3 CASE	Construct
8.1.3.1 Form of the CASE	Construct
8.1.3.2 Execution of a CASE	Construct
8.1.4.1 Form of the DO	Construct
8.1.4.2 Range of a DO	Construct
8.1.4.4 Execution of a DO	Construct
8.1.5 ENABLE	Construct
8.1.5.1 Form of the ENABLE	Construct
8.1.5.2 Execution of an ENABLE	Construct
13.31.1.7.4 Array	Construction Functions
13.6.6 Array	Construction Functions
13.6.6 Array	Construction Functions
13.8.13 Array	Construction Functions
13.8.13 Array	Construction Functions
13.31.1.5 Array	Constructors
4.5 Array	Constructors
8.1.2.3 Examples of IF	Constructs
8.1.3.3 Examples of CASE	Constructs
8.1.4.3 Active and Inactive DO	Constructs
8.1.4.5 Examples of DO	Constructs
8.1.5.5 Examples of ENABLE	Constructs
8.1 Executable	Constructs Containing Blocks
8.1.1.1 Executable	Constructs in Blocks
8.1 Executable Constructs	Containing Blocks
12.5.2.7	CONTAINS Statement
3.3.1.3 Statement	Continuation
8.3	CONTINUE Statement
10.4 Positioning by Format	Control
8 EXECUTION	CONTROL
8.1.4 Iteration	Control
10.6	Control Edit Descriptors
8.1.1.2	Control Flow in Blocks
9.4.1	Control Information List
1.5.4 Text	Conventions
1.5.3 Syntax	Conventions and Characteristics
13.10.5	Core
1.6 Deprecated Features,	Core Conformance
9.4.1.7 Nulls	Count
9.4.1.8 Values	Count
8.1.4.4.2 The Execution	Cycle
8.1.4.4.3	Cycle Interruption
10.5.1.2.2 E and	D Editing
11.3.2.2 Global	Data
Models for Integer and Real	Data 13.5.1
Models for Integer and Real	Data 13.5.1
11.4	Data Abstraction
2.4	Data Concepts
10.5	Data Edit Descriptors
2.4.3	Data Entity
2.4.3.1	Data Object
SPECIFICATIONS 5	DATA OBJECT DECLARATIONS AND
Characteristics of Dummy	Data Objects 12.2.1.1
Arguments Associated with Dummy	Data Objects 12.4.1.1
14.2.2.3 Association of	Data Objects
5.4 Storage Association of	Data Objects
6 USE OF	DATA OBJECTS
5.5	DATA Statement
11.3.2.3	Data Structures
13.16.3 BLOCK	DATA Subprogram
11.5 Block	Data Subprograms
9.2.1.3.1 File Position Prior to	Data Transfer
9.2.1.3.2 File Position After	Data Transfer
9.4.3.1 Direction of	Data Transfer
9.4.3.4	Data Transfer