Reference number of working document: **ISO/IEC JTC1/SC22/WG5 N1578**

Date: 2003-10-8

Reference number of document: **ISO/IEC FCD 1539-1:2004(E)**

Committee identification: ISO/IEC JTC1/SC22

Secretariat: ANSI

Information technology - Programming languages - Fortran - Part 1: Base Language

Technologies de l'information - Langages de programmation - Fortran - Partie 1: Langage de base

Contents

List of Tables

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and nongovernmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication of an International Standard requires approval by at least 75% of the national bodies casting a vote.

International Standard ISO/IEC 1539-1 was prepared by Joint Technical Committee ISO/IEC/JTC1, *Information technology*, Subcommittee SC22, *Programming languages, their environments and system software interfaces*.

This fourth edition cancels and replaces the third edition (ISO/IEC 1539-1:1997), which has been technically revised.

ISO/IEC 1539 consists of the following parts, under the general title *Information technology — Programming languages — Fortran*:

— *Part 1: Base language*

— *Part 2: Varying length character strings*

— *Part 3: Conditional Compilation*

The annexes of this part of ISO/IEC 1539 are for information only.

Introduction

Standard programming language Fortran

This part of the international standard comprises the specification of the base Fortran language, informally known as Fortran 2003. With the limitations noted in 1.6.2, the syntax and semantics of Fortran 95 are contained entirely within Fortran 2003. Therefore, any standard-conforming Fortran 95 program not affected by such limitations is a standard conforming Fortran 2003 program. New features of Fortran 2003 can be compatibly incorporated into such Fortran 95 programs, with any exceptions indicated in the text of this part of the standard.

Fortran 2003 contains several extensions to Fortran 95; among them are:

(1) Derived-type enhancements: parameterized derived types (allows the kind, length, or shape of a derived type's components to be chosen when the derived type is used), mixed component accessibility (allows different components to have different accessibility), public entities of private type, improved structure constructors, and finalizers.

(2) Object oriented programming support: enhanced data abstraction (allows one type to extend the definition of another type), polymorphism (allows the type of a variable to vary at runtime), dynamic type allocation, SELECT TYPE construct (allows a choice of execution flow depending upon the type a polymorphic object currently has), and type-bound procedures.

(3) The ASSOCIATE construct (allows a complex expression or object to be denoted by a simple symbol).

(4) Data manipulation enhancements: allocatable components, deferred type parameters, VOLATILE attribute, explicit type specification in array constructors, INTENT specification of pointer arguments, specified lower bounds of pointer assignment and pointer rank remapping, extended initialization expressions, MAX and MIN intrinsics for character type, and enhanced complex constants.

(5) Input/output enhancements: asynchronous transfer operations (allows a program to continue to process data while an input/output transfer occurs), stream access (allows access to a file without reference to any record structure), user specified transfer operations for derived types, user specified control of rounding during format conversions, the FLUSH statement, named constants for preconnected units, regularization of input/output keywords, and access to input/output error messages.

(6) Procedure pointers.

(7) Scoping enhancements: the ability to rename defined operators (supports greater data abstraction) and control of host association into interface bodies.

(8) Support for IEC 60559 (IEEE 754) exceptions and arithmetic (to the extent a processor's arithmetic supports the IEC standard).

(9) Interoperability with the C programming language (allows portable access to many libraries and the low-level facilities provided by C and allows the portable use of Fortran libraries by programs written in C).

(10) Support for international usage: (ISO 10646) and choice of decimal or comma in numeric formatted input/output.

(11) Enhanced integration with the host operating system: access to command line arguments, environment variables, and the processor's error messages.

Organization of this part of ISO/IEC 1539

This part of ISO/IEC 1539 is organized in 16 sections, dealing with 8 conceptual areas. These 8 areas, and the sections in which they are treated, are:

High/low level concepts	Sections 1, 2, 3
Data concepts	Sections 4, 5, 6
Computations	Sections 7, 13, 14
Execution control	Section 8
Input/output	Sections 9, 10
Program units	Sections 11, 12
Interoperability with C	Section 15
Scoping and association rules	Section 16

It also contains the following nonnormative material:

Glossary	A
Decremental features	B
Extended notes	C
Syntax rules	D
Index	E

Information technology — Programming languages — Fortran —

Part 1:

Base Language

Section 1: Overview

1.1 Scope

ISO/IEC 1539 is a multipart International Standard; the parts are published separately. This publication, ISO/IEC 1539-1, which is the first part, specifies the form and establishes the interpretation of programs expressed in the base Fortran language. The purpose of this part of ISO/IEC 1539 is to promote portability, reliability, maintainability, and efficient execution of Fortran programs for use on a variety of computing systems. The second part, ISO/IEC 1539-2, defines additional facilities for the manipulation of character strings of variable length. The third part, ISO/IEC 1539-3, defines a standard conditional compilation facility for Fortran. A processor conforming to part 1 need not conform to ISO/IEC 1539-2 or ISO/IEC 1539-3; however, conformance to either assumes conformance to this part. Throughout this publication, the term "this standard" refers to ISO/IEC 1539-1.

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

1.4 Exclusions

This standard does not specify

(1) The mechanism by which programs are transformed for use on computing systems,
(2) The operations required for setup and control of the use of programs on computing systems,
(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 this standard fails to establish an interpretation except for the processor detection and reporting requirements in items (2) through (8) of

1.5,

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

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

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

(8) The physical properties and implementation of storage.

1.5 Conformance

A program (2.2.1) is a **standard-conforming program** if it uses only those forms and relationships described herein and if the program has an interpretation according to this standard. A program unit (2.2) conforms to this standard if it can be included in a program in a manner that allows the program to be standard conforming.

A processor conforms to this standard if

(1) It executes any standard-conforming program in a manner that fulfills the interpretations herein, subject to any limits that the processor may impose on the size and complexity of the program;

(2) It contains the capability to detect and report the use within a submitted program unit of a form designated herein as obsolescent, insofar as such use can be detected by reference to the numbered syntax rules and constraints;

(3) It contains the capability to detect and report the use within a submitted program unit of an additional form or relationship that is not permitted by the numbered syntax rules or constraints, including the deleted features described in Annex B;

(4) It contains the capability to detect and report the use within a submitted program unit of an intrinsic type with a kind type parameter value not supported by the processor (4.4);

(5) It contains the capability to detect and report the use within a submitted program unit of source form or characters not permitted by Section 3;

(6) It contains the capability to detect and report the use within a submitted program of name usage not consistent with the scope rules for names, labels, operators, and assignment symbols in Section 16;

(7) It contains the capability to detect and report the use within a submitted program unit of intrinsic procedures whose names are not defined in Section 13; and

(8) It contains the capability to detect and report the reason for rejecting a submitted program.

However, in a format specification that is not part of a FORMAT statement (10.1.1), a processor need not detect or report the use of deleted or obsolescent features, or the use of additional forms or relationships.

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 a 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 is given the EXTERNAL attribute (5.1.2.6) in the scoping unit (16). A standard-conforming program shall not use nonstandard intrinsic procedures or modules that have been added by the processor.

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 program will execute consistently on all or any standard-conforming processors.

In some cases, this standard allows the provision of facilities that are not completely specified in the standard. These facilities are identified as **processor dependent**. They shall be provided, with methods or semantics determined by the processor.

NOTE 1.1

The processor should be accompanied by documentation that specifies the limits it imposes on the size and complexity of a program and the means of reporting when these limits are exceeded, that defines the additional forms and relationships it allows, and that defines the means of reporting the use of additional forms and relationships and the use of deleted or obsolescent forms. In this context, the use of a deleted form is the use of an additional form.

The processor should be accompanied by documentation that specifies the methods or semantics of processor-dependent facilities.

1.6 Compatibility

Each standard since ISO 1539:1980 (informally referred to as FORTRAN 77), defines more intrinsic procedures than the previous one. Therefore, a Fortran program conforming to an older standard may have a different interpretation under a newer standard if it invokes an external procedure having the same name as one of the new standard intrinsic procedures, unless that procedure is specified to have the EXTERNAL attribute.

1.6.1 Fortran 95 compatibility

Except as identified in this section, this standard is an upward compatible extension to the preceding Fortran International Standard, ISO/IEC 1539-1:1997 (Fortran 95). Any standard-conforming Fortran 95 program remains standard-conforming under this standard. The following Fortran 95 features may have different interpretations in this standard:

(1) Earlier Fortran standards had the concept of printing, meaning that column one of formatted output had special meaning for a processor-dependent (possibly empty) set of logical units. This could be neither detected nor specified by a standard-specified means. The interpretation of the first column is not specified by this standard.

(2) This standard specifies a different output format for real zero values in list-directed and namelist output.

(3) If the processor can distinguish between positive and negative real zero, this standard requires different returned values for ATAN2(Y,X) when X $<$ 0 and Y is negative real zero and for LOG(X) and SQRT(X) when X is complex with REAL(X) $<$ 0 and negative zero imaginary part.

1.6.2 Fortran 90 compatibility

Except for the deleted features noted in Annex B.1, and except as identified in this section, this standard is an upward compatible extension to ISO/IEC 1539:1991 (Fortran 90). Any standard-conforming Fortran 90 program that does not use one of the deleted features remains standard-conforming under this standard.

The PAD= specifier in the INQUIRE statement in this standard returns the value UNDEFINED if there is no connection or the connection is for unformatted input/output. Fortran 90 specified YES.

Fortran 90 specified that if the second argument to MOD or MODULO was zero, the result was processor dependent. This standard specifies that the second argument shall not be zero.

1.6.3 Fortran 77 compatibility

Except for the deleted features noted in Annex B.1, and except as identified in this section, this standard is an upward compatible extension to ISO 1539:1980 (Fortran 77). Any standard-conforming Fortran 77 program that does not use one of the deleted features noted in Annex B.1 and that does not depend on the differences specified here remains standard conforming under this standard. This standard restricts the behavior for some features that were processor dependent in Fortran 77. Therefore, a standard-conforming Fortran 77 program that uses one of these processor-dependent features may have a different interpretation under this standard, yet remain a standard-conforming program. The following Fortran 77 features may have different interpretations in this standard:

(1) Fortran 77 permitted a processor to supply more precision derived from a real constant than can be represented in a real datum when the constant is used to initialize a data object of type double precision real in a DATA statement. This standard does not permit a processor this option.

(2) If a named variable that was not in a common block was initialized in a DATA statement and did not have the SAVE attribute specified, Fortran 77 left its SAVE attribute processor dependent. This standard specifies (5.2.5) that this named variable has the SAVE attribute.

(3) Fortran 77 specified that the number of characters required by the input list was to be less than or equal to the number of characters in the record during formatted input. This standard specifies (9.5.3.4.2) that the input record is logically padded with blanks if there are not enough characters in the record, unless the PAD= specifier with the value 'NO' is specified in an appropriate OPEN or READ statement.

(4) A value of 0 for a list item in a formatted output statement will be formatted in a different form for some G edit descriptors. In addition, this standard specifies how rounding of values will affect the output field form, but Fortran 77 did not address this issue. Therefore, some Fortran 77 processors may produce an output form different from the output form produced by Fortran 2003 processors for certain combinations of values and G edit descriptors.

(5) If the processor can distinguish between positive and negative real zero, the behavior of the SIGN intrinsic function when the second argument is negative real zero is changed by this standard.

1.7 Notation used in this standard

In this standard, "shall" is to be interpreted as a requirement; conversely, "shall not" is to be interpreted as a prohibition. Except where stated otherwise, such requirements and prohibitions apply to programs rather than processors.

1.7.1 Informative notes

Informative notes of explanation, rationale, examples, and other material are interspersed with the normative body of this publication. The informative material is nonnormative; it is identified by being in shaded, framed boxes that have numbered headings beginning with "NOTE."

1.7.2 Syntax rules

Syntax rules describe the forms that Fortran lexical tokens, statements, and constructs may take. These syntax rules are expressed in a variation of Backus-Naur form (BNF) in which:

(1) Characters from the Fortran character set (3.1) are interpreted literally as shown, except where otherwise noted.

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

Common abbreviations used in syntactic terms are:

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

(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 that may occur zero or more times
■	continues a syntax rule

(4) Each syntax rule is given a unique identifying number of the form Rsnn, where s is a one- or two-digit section number and nn is a two-digit sequence number within that section. The syntax rules are distributed as appropriate throughout the text, and are referenced by number as needed. Some rules in Sections 2 and 3 are more fully described in later sections; in such cases, the section number s is the number of the later section where the rule is repeated.

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

NOTE 1.2

An example of the use of the syntax rules is:

digit-string **is** *digit* [*digit*] ...

The following are examples of forms for a digit string allowed by the above rule:

digit
digit digit
digit digit digit digit
digit digit digit digit digit digit digit digit

If particular entities are substituted for *digit*, actual digit strings might be:

```
4
67
1999
10243852
```

1.7.3 Constraints

Each constraint is given a unique identifying number of the form Csnn, where s is a one- or two-digit section number and nn is a two-digit sequence number within that section.

Often a constraint is associated with a particular syntax rule. Where that is the case, the constraint is annotated with the syntax rule number in parentheses. A constraint that is associated with a syntax rule constitutes part of the definition of the syntax term defined by the rule. It thus applies in all places where the syntax term appears.

Some constraints are not associated with particular syntax rules. The effect of such a constraint is similar to that of a restriction stated in the text, except that a processor is required to have the capability to detect and report violations of constraints (1.5). In some cases, a broad requirement is stated in text and a subset of the same requirement is also stated as a constraint. This indicates that a standard-conforming program is required to adhere to the broad requirement, but that a standard-conforming processor is required only to have the capability of diagnosing violations of the constraint.

1.7.4 Assumed syntax rules

In order to minimize the number of additional syntax rules and convey appropriate constraint information, the following rules are assumed; an explicit syntax rule for a term overrides an assumed rule. The letters "*xyz*" stand for any syntactic class phrase:

R101	*xyz-list*	**is**	*xyz* [, *xyz*] ...
R102	*xyz-name*	**is**	*name*
R103	*scalar-xyz*	**is**	*xyz*

C101 (R103) *scalar-xyz* shall be scalar.

1.7.5 Syntax conventions and characteristics

(1) Any syntactic class name ending in "*-stmt*" follows the source form statement rules: it shall be delimited by end-of-line or semicolon, and may be labeled unless it forms part of another statement (such as an IF or WHERE statement). Conversely, everything considered to be a source form statement is given a "*-stmt*" ending in the syntax rules.

(2) The rules on statement ordering are described rigorously in the definition of *program-unit* (R202). Expression hierarchy is described rigorously in the definition of *expr* (R722).

(3) The suffix "*-spec*" is used consistently for specifiers, such as input/output statement specifiers. It also is used for type declaration attribute specifications (for example, "*array-spec*" in R510), and in a few other cases.

(4) Where reference is made to a type parameter, including the surrounding parentheses, the suffix "*-selector*" is used. See, for example, "*kind-selector*" (R404) and "*length-selector*" (R425).

(5) The term "*subscript*" (for example, R618, R619, and R620) is used consistently in array definitions.

1.7.6 Text conventions

In the descriptive text, an equivalent English word is frequently used in place of a syntactic term. Particular statements and attributes are identified in the text by an upper-case keyword, e.g., "END statement". Boldface words are used in the text where they are first defined with a specialized meaning. The descriptions of obsolescent features appear in a smaller type size.

NOTE 1.3

This sentence is an example of the type size used for obsolescent features.

1.8 Deleted and obsolescent features

This standard protects the users' investment in existing software by including all but five of the language elements of Fortran 90 that are not processor dependent. This standard identifies two categories of outmoded features. There are five in the first category, **deleted features**, which consists of features considered to have been redundant in FORTRAN 77 and largely unused in Fortran 90. Those in the second category, **obsolescent features**, are considered to have been redundant in Fortran 90 and Fortran 95, but are still frequently used.

1.8.1 Nature of deleted features

(1) Better methods existed in FORTRAN 77.

(2) These features are not included in Fortran 95 or this revision of Fortran.

1.8.2 Nature of obsolescent features

(1) Better methods existed in Fortran 90 and Fortran 95.

(2) It is recommended that programmers use these better methods in new programs and convert existing code to these methods.

(3) These features are identified in the text of this document by a distinguishing type font (1.7.6).

(4) If the use of these features becomes insignificant, future Fortran standards committees should consider deleting them.

(5) The next Fortran standards committee should consider for deletion only those language features that appear in the list of obsolescent features.

(6) Processors supporting the Fortran language should support these features as long as they continue to be used widely in Fortran programs.

1.9 Normative references

The following standards contain provisions which, through reference in this standard, constitute provisions of this standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO/IEC 646:1991, *Information technology—ISO 7-bit coded character set for information interchange*.
ISO/IEC 646:1991 (International Reference Version) is the international equivalent of ANSI X3.4-1986, commonly known as ASCII.

ISO 8601:1988, *Data elements and interchange formats—Information interchange—Representation of dates and times*.

ISO/IEC 9899:1999, *Information technology—Programming languages—C*.
This standard refers to ISO/IEC 9899:1999 as the C standard.

ISO/IEC 10646-1:2000, *Information technology—Universal multiple-octet coded character set (UCS)—Part 1: Architecture and basic multilingual plane*.

IEC 60559 (1989-01), *Binary floating-point arithmetic for microprocessor systems*.
Because IEC 60559 (1989-01) was originally IEEE 754-1985, *Standard for binary floating-point arithmetic*, and is widely known by this name, this standard refers to it as the IEEE standard.

Section 2: Fortran terms and concepts

2.1 High level syntax

This section introduces the terms associated with program units and other Fortran concepts above the construct, statement, and expression levels and illustrates their relationships. The notation used in this standard is described in 1.7.

NOTE 2.1

Constraints and other information related to the rules that do not begin with R2 appear in the appropriate section.

```
R201   program                  is  program-unit
                                        [ program-unit ] ...
```

A *program* shall contain exactly one *main-program program-unit* or a main program defined by means other than Fortran, but not both.

```
R202   program-unit             is  main-program
                                or  external-subprogram
                                or  module
                                or  block-data
R1101  main-program             is  [ program-stmt ]
                                        [ specification-part ]
                                        [ execution-part ]
                                        [ internal-subprogram-part ]
                                        end-program-stmt
R203   external-subprogram      is  function-subprogram
                                or  subroutine-subprogram
R1223  function-subprogram      is  function-stmt
                                        [ specification-part ]
                                        [ execution-part ]
                                        [ internal-subprogram-part ]
                                        end-function-stmt
R1231  subroutine-subprogram    is  subroutine-stmt
                                        [ specification-part ]
                                        [ execution-part ]
                                        [ internal-subprogram-part ]
                                        end-subroutine-stmt
R1104  module                   is  module-stmt
                                        [ specification-part ]
                                        [ module-subprogram-part ]
                                        end-module-stmt
R1116  block-data               is  block-data-stmt
                                        [ specification-part ]
                                        end-block-data-stmt
R204   specification-part       is  [ use-stmt ] ...
                                        [ import-stmt ] ...
                                        [ implicit-part ]
                                        [ declaration-construct ] ...
```

R205	*implicit-part*	**is**	[*implicit-part-stmt*] ...
			implicit-stmt
R206	*implicit-part-stmt*	**is**	*implicit-stmt*
		or	*parameter-stmt*
		or	*format-stmt*
		or	*entry-stmt*
R207	*declaration-construct*	**is**	*derived-type-def*
		or	*entry-stmt*
		or	*enum-def*
		or	*format-stmt*
		or	*interface-block*
		or	*parameter-stmt*
		or	*procedure-declaration-stmt*
		or	*specification-stmt*
		or	*type-declaration-stmt*
		or	*stmt-function-stmt*
R208	*execution-part*	**is**	*executable-construct*
			[*execution-part-construct*] ...
R209	*execution-part-construct*	**is**	*executable-construct*
		or	*format-stmt*
		or	*entry-stmt*
		or	*data-stmt*
R210	*internal-subprogram-part*	**is**	*contains-stmt*
			internal-subprogram
			[*internal-subprogram*] ...
R211	*internal-subprogram*	**is**	*function-subprogram*
		or	*subroutine-subprogram*
R1107	*module-subprogram-part*	**is**	*contains-stmt*
			module-subprogram
			[*module-subprogram*] ...
R1108	*module-subprogram*	**is**	*function-subprogram*
		or	*subroutine-subprogram*
R212	*specification-stmt*	**is**	*access-stmt*
		or	*allocatable-stmt*
		or	*asynchronous-stmt*
		or	*bind-stmt*
		or	*common-stmt*
		or	*data-stmt*
		or	*dimension-stmt*
		or	*equivalence-stmt*
		or	*external-stmt*
		or	*intent-stmt*
		or	*intrinsic-stmt*
		or	*namelist-stmt*
		or	*optional-stmt*
		or	*pointer-stmt*
		or	*protected-stmt*
		or	*save-stmt*
		or	*target-stmt*
		or	*volatile-stmt*
		or	*value-stmt*
R213	*executable-construct*	**is**	*action-stmt*
		or	*associate-construct*
		or	*case-construct*

or *do-construct*
or *forall-construct*
or *if-construct*
or *select-type-construct*
or *where-construct*

R214 *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 *end-function-stmt*
or *end-program-stmt*
or *end-subroutine-stmt*
or *exit-stmt*
or *flush-stmt*
or *forall-stmt*
or *goto-stmt*
or *if-stmt*
or *inquire-stmt*
or *nullify-stmt*
or *open-stmt*
or *pointer-assignment-stmt*
or *print-stmt*
or *read-stmt*
or *return-stmt*
or *rewind-stmt*
or *stop-stmt*
or *wait-stmt*
or *where-stmt*
or *write-stmt*
or *arithmetic-if-stmt*
or *computed-goto-stmt*

C201 (R208) An *execution-part* shall not contain an *end-function-stmt*, *end-program-stmt*, or *end-subroutine-stmt*.

2.2 Program unit concepts

Program units are the fundamental components of a Fortran program. A **program unit** may be a main program, an external subprogram, a module, or a block data program unit. A subprogram may be a function subprogram or a subroutine subprogram. A module contains definitions that are to be made accessible to other program units. A block data program unit is used to specify initial values for data objects in named common blocks. Each type of program unit is described in Section 11 or 12. An **external subprogram** is a subprogram that is not in a main program, a module, or another subprogram. An **internal subprogram** is a subprogram that is in a main program or another subprogram. A **module subprogram** is a subprogram that is in a module but is not an internal subprogram.

A program unit consists of a set of nonoverlapping scoping units. A **scoping unit** is

(1) A program unit or subprogram, excluding any scoping units in it,
(2) A derived-type definition (4.5.1), or
(3) An interface body, excluding any scoping units in it.

A scoping unit that immediately surrounds another scoping unit is called the **host scoping unit** (often abbreviated to **host**).

2.2.1 Program

A **program** consists of exactly one main program, any number (including zero) of other kinds of program units, and any number (including zero) of external procedures and other entities defined by means other than Fortran.

NOTE 2.2

There is a restriction that there shall be no more than one unnamed block data program unit (11.3).

This standard places no ordering requirement on the program units that constitute a program, but because the public portions of a module are required to be available by the time a module reference (11.2.1) is processed, a processor may require a particular order of processing of the program units.

2.2.2 Main program

The Fortran main program is described in 11.1.

2.2.3 Procedure

A **procedure** encapsulates an arbitrary sequence of actions that may be invoked directly during program execution. Procedures are either functions or subroutines. A **function** is a procedure that is invoked in an expression; its invocation causes a value to be computed which is then used in evaluating the expression. The variable that returns the value of a function is called the **result variable**. A **subroutine** is a procedure that is invoked in a CALL statement, by a defined assignment statement, or by some operations on derived-type entities. Unless it is a pure procedure, a subroutine may be used to change the program state by changing the values of any of the data objects accessible to the subroutine; unless it is a pure procedure, a function 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 procedure that is defined by an external subprogram or by means other than Fortran. An external procedure may be invoked by the main program or by any procedure of a program.

2.2.3.2 Module procedure

A **module procedure** is a procedure that is defined by a module subprogram (R1108). The module containing the subprogram is the **host** scoping unit of the module procedure.

2.2.3.3 Internal procedure

An **internal procedure** is a procedure that is defined by an internal subprogram (R211). The containing main program or subprogram is the **host** scoping unit of the internal procedure. An internal procedure is local to its host in the sense that the internal procedure is accessible within the host scoping unit and all its other internal procedures but is not accessible elsewhere.

2.2.3.4 Interface block

An **interface body** describes an abstract interface or the interface of a dummy procedure, external procedure, procedure pointer, or type-bound procedure.

An **interface block** is a specific interface block, an abstract interface block, or a generic interface block. A specific interface block is a collection of interface bodies. A generic interface block may also be used to specify that procedures may be invoked

(1) By using a generic name,
(2) By using a defined operator,
(3) By using a defined assignment, or
(4) For derived-type input/output.

2.2.4 Module

A **module** contains (or accesses from other modules) definitions that are to be made accessible to other program units. These definitions include data object declarations, type definitions, procedure definitions, and interface blocks. A scoping unit in another program unit may access the definitions in a module. Modules are further described in Section 11.

2.3 Execution concepts

Each Fortran statement is classified as either an executable statement or a nonexecutable statement. There are restrictions on the order in which statements may appear in a program unit, and not all executable statements may appear in all contexts.

2.3.1 Executable/nonexecutable statements

Program execution is a sequence, in time, of 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 behavior of the program unit. The executable statements are all of those that make up the syntactic class *executable-construct*.

Nonexecutable statements do not specify actions; they are used to configure the program environment in which actions take place. The nonexecutable statements are all those not classified as executable.

2.3.2 Statement order

The syntax rules of subclause 2.1 specify the statement order within program units and subprograms. These rules are illustrated in Table 2.1 and Table 2.2. Table 2.1 shows the ordering rules for statements and applies to all program units, subprograms, and interface bodies. Vertical lines delineate varieties of statements that may be interspersed and horizontal lines delineate varieties of statements that shall not be interspersed. Internal or module subprograms shall follow a CONTAINS statement. Between USE and CONTAINS statements in a subprogram, nonexecutable statements generally precede executable statements, although the ENTRY statement, FORMAT statement, and DATA statement may appear among the executable statements. Table 2.2 shows which statements are allowed in a scoping unit.

Table 2.1: **Requirements on statement ordering**

<table>
<tr><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE,
MODULE, or BLOCK DATA statement</td></tr>
<tr><td colspan="3">USE statements</td></tr>
<tr><td colspan="3">IMPORT statements</td></tr>
<tr><td rowspan="4">FORMAT
and
ENTRY
statements</td><td colspan="2">IMPLICIT NONE</td></tr>
<tr><td>PARAMETER
statements</td><td>IMPLICIT
statements</td></tr>
<tr><td>PARAMETER
and DATA
statements</td><td>Derived-type definitions,
interface blocks,
type declaration statements,
enumeration declarations,
procedure declarations,
specification statements,
and statement function statements</td></tr>
<tr><td>DATA
statements</td><td>Executable
constructs</td></tr>
<tr><td colspan="3">CONTAINS statement</td></tr>
<tr><td colspan="3">Internal subprograms
or module subprograms</td></tr>
<tr><td colspan="3">END statement</td></tr>
</table>

Table 2.2: **Statements allowed in scoping units**

Kind of scoping unit:	Main program	Module	Block data	External subprog	Module subprog	Internal subprog	Interface body
USE statement	Yes	Yes	Yes	Yes	Yes	Yes	Yes
IMPORT statement	No	No	No	No	No	No	Yes
ENTRY statement	No	No	No	Yes	Yes	No	No
FORMAT statement	Yes	No	No	Yes	Yes	Yes	No
Misc. decls (see note)	Yes	Yes	Yes	Yes	Yes	Yes	Yes
DATA statement	Yes	Yes	Yes	Yes	Yes	Yes	No
Derived-type definition	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Interface block	Yes	Yes	No	Yes	Yes	Yes	Yes
Executable statement	Yes	No	No	Yes	Yes	Yes	No
CONTAINS statement	Yes	Yes	No	Yes	Yes	No	No
Statement function statement	Yes	No	No	Yes	Yes	Yes	No

Notes for Table 2.2:

1) Misc. declarations are PARAMETER statements, IMPLICIT statements, type declaration statements, enum statements, procedure declaration statements, and specification statements.

2) The scoping unit of a module does not include any module subprograms that the module contains.

2.3.3 The END statement

An *end-program-stmt*, *end-function-stmt*, *end-subroutine-stmt*, *end-module-stmt*, or *end-block-data-stmt* is an **END statement**. Each program unit, module subprogram, and internal subprogram shall have exactly one END statement. The *end-program-stmt*, *end-function-stmt*, and *end-subroutine-stmt* statements are executable, and may be branch target statements (8.2). Executing an *end-program-stmt* causes normal termination of execution of the program. Executing an *end-function-stmt* or *end-subroutine-stmt* is equivalent to executing a *return-stmt* with no *scalar-int-expr*.

The *end-module-stmt* and *end-block-data-stmt* statements are nonexecutable.

2.3.4 Execution sequence

If a program contains a Fortran main program, execution of the program begins with the first executable construct of the main program. The execution of a main program or subprogram involves execution of the executable constructs within its scoping unit. When a procedure is invoked, execution begins with the first executable construct appearing after the invoked entry point. With the following exceptions, the effect of execution is as if the executable constructs are executed in the order in which they appear in the main program or subprogram until a STOP, RETURN, or END statement is executed. The exceptions are the following:

(1) Execution of a branching statement (8.2) changes the execution sequence. These statements explicitly specify a new starting place for the execution sequence.

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

(3) END=, ERR=, and EOR= specifiers may result in a branch.

(4) Alternate returns may result in a branch.

Internal subprograms may precede the END statement of a main program or a subprogram. The execution sequence excludes all such definitions.

Normal termination of execution of the program occurs if a STOP statement or *end-program-stmt* is executed. Normal termination of execution of a program also may occur during execution of a procedure defined by a companion processor (C standard 5.1.2.2.3 and 7.20.4.3). If normal termination of execution occurs within a Fortran program unit and the program incorporates procedures defined by a companion processor, the process of execution termination shall include the effect of executing the C exit() function (C standard 7.20.4.3).

2.4 Data concepts

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

2.4.1 Type

A **type** is a named category of data that is characterized by a set of values, a syntax for denoting these values, and a set of operations that interpret and manipulate the values. This central concept is described in 4.1.

A type may be parameterized, in which case the set of data values, the syntax for denoting them, and the set of operations depend on the values of one or more parameters. Such a parameter is called a **type parameter** (4.2).

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

2.4.1.1 Intrinsic type

An **intrinsic type** is a type that is defined by the language, along with operations, and is always accessible. The intrinsic types are integer, real, complex, character, and logical. The properties of intrinsic types are described in 4.4. The intrinsic type parameters are KIND and LEN.

The **kind type parameter** indicates the decimal exponent range for the integer type (4.4.1), the decimal precision and exponent range for the real and complex types (4.4.2, 4.4.3), and the representation methods for the character and logical types (4.4.4, 4.4.5). The **character length parameter** specifies the number of characters for the character type.

2.4.1.2 Derived type

A **derived type** is a type that is not defined by the language but requires a type definition to declare its components. A scalar object of such a derived type is called a **structure** (5.1.1.1). Derived types may be parameterized. Assignment of structures is defined intrinsically (7.4.1.3), but there are no intrinsic operations for structures. For each derived type, a structure constructor is available to provide values (4.5.9). In addition, data objects of derived type may be used as procedure arguments and function results, and may appear in input/output lists. If additional operations are needed for a derived type, they shall be supplied as procedure definitions.

Derived types are described further in 4.5.

2.4.2 Data value

Each intrinsic type has associated with it a set of values that a datum of that type may take, depending on the values of the type parameters. The values for each intrinsic type are described in 4.4. The values that objects of a derived type may assume are determined by the type definition, type parameter values, and the sets of values of its components.

2.4.3 Data entity

A **data entity** is a data object, the result of the evaluation of an expression, or the result of the execution of a function reference (called the function result). A data entity has a type and type parameters; it may have a data value (an exception is an undefined variable). Every data entity has a rank and is thus either a scalar or an array.

2.4.3.1 Data object

A **data object** (often abbreviated to **object**) is a constant (4.1.2), a variable (6), or a subobject of a constant. The type and type parameters of a named data object may be specified explicitly (5.1) or implicitly (5.3).

Subobjects are portions of certain objects that may be referenced and defined (variables only) independently of the other portions. These include portions of arrays (array elements and array sections), portions of character strings (substrings), portions of complex objects (real and imaginary parts), and portions of structures (components). Subobjects are themselves data objects, but subobjects are referenced only by object designators or intrinsic functions. A subobject of a variable is a variable. Subobjects are described in Section 6.

Objects referenced by a name are:

a named scalar	(a scalar object)
a named array	(an array object)

Subobjects referenced by an object designator are:

an array element	(a scalar subobject)
an array section	(an array subobject)
a structure component	(a scalar or an array subobject)
a substring	(a scalar subobject)

Subobjects of complex objects may also be referenced by intrinsic functions.

2.4.3.1.1 Variable

A **variable** may have a value and may be defined and redefined during execution of a program.

A named **local variable** of the scoping unit of a module, main program, or subprogram, is a named variable that is a local entity of the scoping unit, is not a dummy argument, is not in COMMON, does not have the BIND attribute, and is not accessed by use or host association. A subobject of a named local variable is also a local variable.

2.4.3.1.2 Constant

A **constant** has a value and cannot become defined, redefined, or undefined during execution of a program. A constant with a name is called a **named constant** and has the PARAMETER attribute (5.1.2.10). A constant without a name is called a **literal constant** (4.4).

2.4.3.1.3 Subobject of a constant

A **subobject of a constant** is a portion of a constant. The portion referenced may depend on the value of a variable.

NOTE 2.3

For example, given:

```
CHARACTER (LEN = 10), PARAMETER :: DIGITS = '0123456789'
CHARACTER (LEN = 1)             :: DIGIT
INTEGER :: I
   ...
DIGIT = DIGITS (I:I)
```

DIGITS is a named constant and DIGITS (I:I) designates a subobject of the constant DIGITS.

2.4.3.2 Expression

An **expression** (7.1) produces a data entity when evaluated. An expression represents either a data reference or a computation; it is formed from operands, operators, and parentheses. The type, type parameters, value, and rank of an expression result are determined by the rules in Section 7.

2.4.3.3 Function reference

A **function reference** (12.4.2) produces a data entity when the function is executed during expression evaluation. The type, type parameters, and rank of a function result are determined by the interface of the function (12.2.2). The value of a function result is determined by execution of the function.

2.4.4 Scalar

A **scalar** is a datum that is not an array. Scalars may be of any intrinsic type or derived type.

NOTE 2.4

A structure is scalar even if it has arrays as components.

The **rank** of a scalar is zero. The shape of a scalar is represented by a rank-one array of size zero.

2.4.5 Array

An **array** is a set of scalar 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 and is a scalar. An **array section** is a subset of the elements of an array and is itself an array.

An array may have up to seven dimensions, and any **extent** (number of elements) in any dimension. The **rank** of the array is the number of dimensions; its **size** is the total number of elements, which is equal to the product of the extents. An array may have zero size. The **shape** of an array is determined by its rank and its extent in each dimension, and may be represented as a rank-one array whose elements are the extents. All named arrays shall be declared, and the rank of a named array is specified in its declaration. The rank of a named array, once declared, is constant; the extents may be constant or may vary during execution.

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

A rank-one array may be constructed from scalars and other arrays and may be reshaped into any allowable array shape (4.7).

Arrays may be of any intrinsic type or derived type and are described further in 6.2.

2.4.6 Pointer

A **data pointer** is a data entity that has the POINTER attribute. A **procedure pointer** is a procedure entity that has the POINTER attribute. A **pointer** is either a data pointer or a procedure pointer.

A pointer is **associated** with a **target** by pointer assignment (7.4.2). A data pointer may also be associated with a target by allocation (6.3.1). A pointer is **disassociated** following execution of a NULLIFY statement, following pointer assignment with a disassociated pointer, by default initialization, or by explicit initialization. A data pointer may also be disassociated by execution of a DEALLOCATE statement. A disassociated pointer is not associated with a target (16.4.2).

A pointer that is not associated shall not be referenced or defined.

If a data pointer is an array, the rank is declared, but the extents are determined when the pointer is associated with a target.

2.4.7 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 shall observe the storage restrictions described in 16.4.3.

2.5 Fundamental terms

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

2.5.1 Name and designator

A **name** is used to identify a program constituent, such as a program unit, named variable, named constant, dummy argument, or derived type. The rules governing the construction of names are given in 3.2.1. A **designator** is a name followed by zero or more component selectors, array section selectors, array element selectors, and substring selectors.

An **object designator** is a designator for a data object. A **procedure designator** is a designator for a procedure.

NOTE 2.5

An object name is a special case of an object designator.

2.5.2 Keyword

The term **keyword** is used in two ways.

(1) It is used to describe a word that is part of the syntax of a statement. These keywords are not reserved words; that is, names with the same spellings are allowed. In the syntax rules, such keywords appear literally. In descriptive text, this meaning is denoted by the term "keyword" without any modifier. Examples of statement keywords are: IF, READ, UNIT, KIND, and INTEGER.

(2) It is used to denote names that identify items in a list. In actual argument lists, type parameter lists, and structure constructors, items may be identified by a preceding *keyword* = rather than their position within the list. An **argument keyword** is the name of a dummy argument in the interface for the procedure being referenced, a **type parameter keyword** is the name of a type parameter in the type being specified, and a **component keyword** is the name of a component in a structure constructor.

R215 *keyword* **is** *name*

NOTE 2.6

Use of keywords rather than position to identify items in a list can make such lists more readable and allows them to be reordered. This facilitates specification of a list in cases where optional items are omitted.

2.5.3 Association

Association is name association (16.4.1), pointer association (16.4.2), storage association (16.4.3), or inheritance association (16.4.4). Name association is argument association, host association, use association, linkage association, or construct association.

Storage association causes different entities to use the same storage. Any association permits an entity to be identified by different names in the same scoping unit or by the same name or different names in different scoping units.

2.5.4 Declaration

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

2.5.5 Definition

The term **definition** is used in two ways.

(1) It refers to the specification of derived types and procedures.

(2) When an object is given a valid value during program execution, it is said to become **defined**. This is often accomplished by execution of an assignment or input statement. When a variable does not have a predictable value, it is said to be **undefined**. Similarly, when a pointer is associated with a target or nullified, its pointer association status is said to become **defined**. When the association status of a pointer is not predictable, its pointer association status is said to be **undefined**.

Section 16 describes the ways in which variables may become defined and undefined.

2.5.6 Reference

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

A **procedure reference** is the appearance of the procedure designator, operator symbol, or assignment symbol in a context requiring execution of the procedure at that point. An occurrence of user-defined derived-type input/output (10.6.5) or derived-type finalization (4.5.5.1) is also a procedure reference.

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

A **module reference** is the appearance of a module name in a USE statement (11.2.1).

2.5.7 Intrinsic

The qualifier **intrinsic** has two meanings.

(1) The qualifier signifies that the term to which it is applied is defined in this standard. Intrinsic applies to types, procedures, modules, assignment statements, and operators. All intrinsic types, procedures, assignments, and operators may be used in any scoping unit without further definition or specification. Intrinsic modules may be accessed by use association. Intrinsic procedures and modules defined in this standard are called standard intrinsic procedures and standard intrinsic modules, respectively.

(2) The qualifier applies to procedures or modules that are provided by a processor but are not defined in this standard (13, 14, 15.1). Such procedures and modules are called nonstandard intrinsic procedures and nonstandard intrinsic modules, respectively.

2.5.8 Operator

An **operator** specifies a computation involving one (unary operator) or two (binary operator) data values (**operands**). This standard specifies 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 be defined within a program (7.1.3).

2.5.9 Sequence

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

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

2.5.10 Companion processors

A processor has one or more companion processors. A **companion processor** is a processor-dependent mechanism by which global data and procedures may be referenced or defined. A companion processor may be a mechanism that references and defines such entities by a means other than Fortran (12.5.3), it may be the Fortran processor itself, or it may be another Fortran processor. If there is more than one companion processor, the means by which the Fortran processor selects among them are processor dependent.

If a procedure is defined by means of a companion processor that is not the Fortran processor itself, this standard refers to the C function that defines the procedure, although the procedure need not be defined by means of the C programming language.

NOTE 2.7

A companion processor might or might not be a mechanism that conforms to the requirements of the C standard.

For example, a processor may allow a procedure defined by some language other than Fortran or C to be linked (12.5.3) with a Fortran procedure if it can be described by a C prototype as defined in 6.5.5.3 of the C standard.

Section 3: Characters, lexical tokens, and source form

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

3.1 Processor character set

The processor character set is processor dependent. The structure of a processor character set is:

(1) **Control characters**

(2) **Graphic characters**

(a) Letters (3.1.1)

(b) Digits (3.1.2)

(c) Underscore (3.1.3)

(d) Special characters (3.1.4)

(e) Other characters (3.1.5)

The letters, digits, underscore, and special characters make up the **Fortran character set**.

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

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

The default character type shall support a character set that includes the Fortran character set. By supplying nondefault character types, the processor may support additional character sets. The characters available in the ASCII and ISO 10646 character sets are specified by ISO/IEC 646:1991 (International Reference Version) and ISO/IEC 10646-1:2000 UCS-4, respectively; the characters available in other nondefault character types are not specified by this standard, except that one character in each nondefault character type shall be designated as a blank character to be used as a padding character.

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

The set of letters defines the syntactic class *letter*. The processor character set shall include lower-case and upper-case letters. A lower-case letter is equivalent to the corresponding upper-case letter in program units except in a character context (3.3).

NOTE 3.1

The following statements are equivalent:

```
CALL BIG_COMPLEX_OPERATION (NDATE)
call big_complex_operation (ndate)
```

NOTE 3.1 (cont.)

```
Call Big_Complex_Operation (NDate)
```

3.1.2 Digits

The ten **digits** are:

0 1 2 3 4 5 6 7 8 9

The ten digits define the syntactic class *digit*.

3.1.3 Underscore

R303 *underscore* **is** _

The underscore may be used as a significant character in a name.

3.1.4 Special characters

The **special characters** are shown in Table 3.1.

Table 3.1: **Special characters**

<table>
<tr><th>Character</th><th>Name of character</th><th>Character</th><th>Name of character</th></tr>
<tr><td></td><td>Blank</td><td>;</td><td>Semicolon</td></tr>
<tr><td>=</td><td>Equals</td><td>!</td><td>Exclamation point</td></tr>
<tr><td>+</td><td>Plus</td><td>"</td><td>Quotation mark or quote</td></tr>
<tr><td>-</td><td>Minus</td><td>%</td><td>Percent</td></tr>
<tr><td>*</td><td>Asterisk</td><td>&</td><td>Ampersand</td></tr>
<tr><td>/</td><td>Slash</td><td>~</td><td>Tilde</td></tr>
<tr><td>\</td><td>Backslash</td><td><</td><td>Less than</td></tr>
<tr><td>(</td><td>Left parenthesis</td><td>></td><td>Greater than</td></tr>
<tr><td>)</td><td>Right parenthesis</td><td>?</td><td>Question mark</td></tr>
<tr><td>[</td><td>Left square bracket</td><td>'</td><td>Apostrophe</td></tr>
<tr><td>]</td><td>Right square bracket</td><td>`</td><td>Grave accent</td></tr>
<tr><td>{</td><td>Left curly bracket</td><td>^</td><td>Circumflex accent</td></tr>
<tr><td>}</td><td>Right curly bracket</td><td>|</td><td>Vertical line</td></tr>
<tr><td>,</td><td>Comma</td><td>$</td><td>Currency symbol</td></tr>
<tr><td>.</td><td>Decimal point or period</td><td>#</td><td>Number sign</td></tr>
<tr><td>:</td><td>Colon</td><td>@</td><td>Commercial at</td></tr>
</table>

The special characters define the syntactic class *special-character*. Some of the special characters are used for operator symbols, bracketing, and various forms of separating and delimiting other lexical tokens.

3.1.5 Other characters

Additional characters may be representable in the processor, but may appear only in comments (3.3.1.1, 3.3.2.1), character constants (4.4.4), input/output records (9.1.1), and character string edit descriptors (10.2.1).

3.2 Low-level syntax

The **low-level syntax** describes the fundamental lexical tokens of a program unit. **Lexical tokens** are sequences of characters that constitute the building blocks of a program. They are keywords, names, literal constants other than complex literal constants, operators, labels, delimiters, comma, =, =>, :, ::, ;, and %.

3.2.1 Names

Names are used for various entities such as variables, program units, dummy arguments, named constants, and derived types.

R304 *name* **is** *letter* [*alphanumeric-character*] ...

C301 (R304) The maximum length of a *name* is 63 characters.

NOTE 3.2

Examples of names:

```
A1
NAME_LENGTH        (single underscore)
S_P_R_E_A_D__O_U_T (two consecutive underscores)
TRAILER_           (trailing underscore)
```

NOTE 3.3

The word "name" always denotes this particular syntactic form. The word "identifier" is used where entities may be identified by other syntactic forms or by values; its particular meaning depends on the context in which it is used.

3.2.2 Constants

R305 *constant* **is** *literal-constant*
or *named-constant*
R306 *literal-constant* **is** *int-literal-constant*
or *real-literal-constant*
or *complex-literal-constant*
or *logical-literal-constant*
or *char-literal-constant*
or *boz-literal-constant*
R307 *named-constant* **is** *name*
R308 *int-constant* **is** *constant*

C302 (R308)*int-constant* shall be of type integer.

R309 *char-constant* **is** *constant*

C303 (R309) *char-constant* shall be of type character.

3.2.3 Operators

R310 *intrinsic-operator* **is** *power-op*
or *mult-op*
or *add-op*
or *concat-op*

		or	*rel-op*
		or	*not-op*
		or	*and-op*
		or	*or-op*
		or	*equiv-op*
R707	*power-op*	is	**
R708	*mult-op*	is	*
		or	/
R709	*add-op*	is	+
		or	−
R711	*concat-op*	is	//
R713	*rel-op*	is	.EQ.
		or	.NE.
		or	.LT.
		or	.LE.
		or	.GT.
		or	.GE.
		or	==
		or	/=
		or	<
		or	<=
		or	>
		or	>=
R718	*not-op*	is	.NOT.
R719	*and-op*	is	.AND.
R720	*or-op*	is	.OR.
R721	*equiv-op*	is	.EQV.
		or	.NEQV.
R311	*defined-operator*	is	*defined-unary-op*
		or	*defined-binary-op*
		or	*extended-intrinsic-op*
R703	*defined-unary-op*	is	. *letter* [*letter*] ... .
R723	*defined-binary-op*	is	. *letter* [*letter*] ... .
R312	*extended-intrinsic-op*	is	*intrinsic-operator*

3.2.4 Statement labels

A **statement label** provides a means of referring to an individual statement.

R313 *label* **is** *digit* [*digit* [*digit* [*digit* [*digit*]]]]

C304 (R313) At least one digit in a *label* shall be nonzero.

If a statement is labeled, the statement shall contain a nonblank character. The same statement label shall not be given to more than one statement in a scoping unit. Leading zeros are not significant in distinguishing between statement labels.

NOTE 3.4

For example:

```
99999
10
 010
```

NOTE 3.4 (cont.)

are all statement labels. The last two are equivalent.

There are 99999 unique statement labels and a processor shall accept any of them as a statement label. However, a processor may have an implementation limit on the total number of unique statement labels in one program unit.

Any statement may have a statement label, but the labels are used only in the following ways:

(1) The label on a branch target statement (8.2) is used to identify that statement as the possible destination of a branch.

(2) The label on a FORMAT statement (10.1.1) is used to identify that statement as the format specification for a data transfer statement (9.5).

(3) In some forms of the DO construct (8.1.6), the range of the DO construct is identified by the label on the last statement in that range.

3.2.5 Delimiters

Delimiters are used to enclose syntactic lists. The following pairs are delimiters:

`( ... )`

`/ ... /`

`[ ... ]`

`(/ ... /)`

3.3 Source form

A Fortran program unit is a sequence of one or more lines, organized as Fortran statements, comments, and INCLUDE lines. A **line** is a sequence of zero or more characters. Lines following a program unit END statement are not part of that program unit. A Fortran **statement** is a sequence of one or more complete or partial lines.

A **character context** means characters within a character literal constant (4.4.4) or within a character string edit descriptor (10.2.1).

A comment may contain any character that may occur in any character context.

There are two **source forms**: free and fixed. Free form and fixed form shall not be mixed in the same program unit. The means for specifying the source form of a program unit are processor dependent.

3.3.1 Free source form

In **free source form** there are no restrictions on where a statement (or portion of a statement) may appear within a line. A line may contain zero characters. If a line consists entirely of characters of default kind (4.4.4), it may contain at most 132 characters. If a line contains any character that is not of default kind, the maximum number of characters allowed on the line is processor dependent.

Blank characters shall not appear within lexical tokens other than in a character context or in a format specification. Blanks may be inserted freely between tokens to improve readability; for example, blanks may occur between the tokens that form a complex literal constant. A sequence of blank characters outside of a character context is equivalent to a single blank character.

A blank shall be used to separate names, constants, or labels from adjacent keywords, names, constants, or labels.

NOTE 3.5

For example, the blanks after REAL, READ, 30, and DO are required in the following:

```
REAL X
READ 10
30 DO K=1,3
```

One or more blanks shall be used to separate adjacent keywords except in the following cases, where blanks are optional:

Adjacent keywords where separating blanks are optional	
BLOCK DATA	DOUBLE PRECISION
ELSE IF	ELSE WHERE
END ASSOCIATE	END BLOCK DATA
END DO	END ENUM
END FILE	END FORALL
END FUNCTION	END IF
END INTERFACE	END MODULE
END PROGRAM	END SELECT
END SUBROUTINE	END TYPE
END WHERE	GO TO
IN OUT	SELECT CASE
SELECT TYPE	

3.3.1.1 Free form commentary

The character "!" initiates a **comment** except where it appears within a character context. The comment extends to the end of the line. If the first nonblank character on a line is an "!", the line is a comment line. Lines containing only blanks or containing no characters are also comment lines. Comments may appear anywhere in a program unit and may precede the first statement of a program unit or may follow the last statement of a program unit. Comments have no effect on the interpretation of the program unit.

NOTE 3.6

The standard does not restrict the number of consecutive comment lines.

3.3.1.2 Free form statement continuation

The character "&" is used to indicate that the current statement is continued on the next line that is not a comment line. Comment lines cannot be continued; an "&" in a comment has no effect. Comments may occur within a continued statement. When used for continuation, the "&" is not part of the statement. No line shall contain a single "&" as the only nonblank character or as the only nonblank character before an "!" that initiates a comment.

If a noncharacter context is to be continued, an "&" shall be the last nonblank character on the line, or the last nonblank character before an "!". There shall be a later line that is not a comment; the statement is continued on the next such line. If the first nonblank character on that line is an "&", the statement continues at the next character position following that "&"; otherwise, it continues with the first character position of that line.

If a lexical token is split across the end of a line, the first nonblank character on the first following noncomment line shall be an "&" immediately followed by the successive characters of the split token.

If a character context is to be continued, an "&" shall be the last nonblank character on the line and shall not be followed by commentary. There shall be a later line that is not a comment; an "&" shall be the first nonblank character on the next such line and the statement continues with the next character following that "&".

3.3.1.3 Free form statement termination

If a statement is not continued, a comment or the end of the line terminates the statement.

A statement may alternatively be terminated by a ";" character that appears other than in a character context or in a comment. The ";" is not part of the statement. After a ";" terminator, another statement may appear on the same line, or begin on that line and be continued. A ";" shall not appear as the first nonblank character on a line. A sequence consisting only of zero or more blanks and one or more ";" terminators, in any order, is equivalent to a single ";" terminator.

3.3.1.4 Free form statements

A label may precede any statement not forming part of another statement.

NOTE 3.7

No Fortran statement begins with a digit.

A statement shall not have more than 255 continuation lines.

3.3.2 Fixed source form

In **fixed source form**, there are restrictions on where a statement may appear within a line. If a source line contains only default kind characters, it shall contain exactly 72 characters; otherwise, its maximum number of characters is processor dependent.

Except in a character context, blanks are insignificant and may be used freely throughout the program.

3.3.2.1 Fixed form commentary

The character "!" initiates a **comment** except where it appears within a character context or in character position 6. The comment extends to the end of the line. If the first nonblank character on a line is an "!" in any character position other than character position 6, the line is a comment line. Lines beginning with a "C" or "*" in character position 1 and lines containing only blanks are also comment lines. Comments may appear anywhere in a program unit and may precede the first statement of the program unit or may follow the last statement of a program unit. Comments have no effect on the interpretation of the program unit.

NOTE 3.8

The standard does not restrict the number of consecutive comment lines.

3.3.2.2 Fixed form statement continuation

Except within commentary, character position 6 is used to indicate continuation. If character position 6 contains a blank or zero, the line is the initial line of a new statement, which begins in character position 7. If character position 6 contains any character other than blank or zero, character positions 7–72 of the line constitute a continuation of the preceding noncomment line.

NOTE 3.9

An "!" or ";" in character position 6 is interpreted as a continuation indicator unless it appears within commentary indicated by a "C" or "*" in character position 1 or by an "!" in character positions 1–5.

Comment lines cannot be continued. Comment lines may occur within a continued statement.

3.3.2.3 Fixed form statement termination

If a statement is not continued, a comment or the end of the line terminates the statement.

A statement may alternatively be terminated by a ";" character that appears other than in a character context, in a comment, or in character position 6. The ";" is not part of the statement. After a ";" terminator, another statement may begin on the same line, or begin on that line and be continued. A ";" shall not appear as the first nonblank character on a line, except in character position 6. A sequence consisting only of zero or more blanks and one or more ";" terminators, in any order, is equivalent to a single ";" terminator.

3.3.2.4 Fixed form statements

A label, if present, shall occur in character positions 1 through 5 of the first line of a statement; otherwise, positions 1 through 5 shall be blank. Blanks may appear anywhere within a label. A statement following a ";" on the same line shall not be labeled. Character positions 1 through 5 of any continuation lines shall be blank. A statement shall not have more than 255 continuation lines. The program unit END statement shall not be continued. A statement whose initial line appears to be a program unit END statement shall not be continued.

3.4 Including source text

Additional text may be incorporated into the source text of a program unit during processing. This is accomplished with the **INCLUDE line**, which has the form

INCLUDE *char-literal-constant*

The *char-literal-constant* shall not have a kind type parameter value that is a *named-constant*.

An INCLUDE line is not a Fortran statement.

An INCLUDE line shall appear on a single source line where a statement may appear; it shall be the only nonblank text on this line other than an optional trailing comment. Thus, a statement label is not allowed.

The effect of the INCLUDE line is as if the referenced source text physically replaced the INCLUDE line prior to program processing. Included text may contain any source text, including additional INCLUDE lines; such nested INCLUDE lines are similarly replaced with the specified source text. The maximum depth of nesting of any nested INCLUDE lines is processor dependent. Inclusion of the source text referenced by an INCLUDE line shall not, at any level of nesting, result in inclusion of the same source text.

When an INCLUDE line is resolved, the first included statement line shall not be a continuation line and the last included statement line shall not be continued.

The interpretation of *char-literal-constant* is processor dependent. An example of a possible valid interpretation is that *char-literal-constant* is the name of a file that contains the source text to be included.

NOTE 3.10

In some circumstances, for example where source code is maintained in an INCLUDE file for use in programs whose source form might be either fixed or free, observing the following rules allows the code to be used with either source form:

(1) Confine statement labels to character positions 1 to 5 and statements to character positions 7 to 72;

(2) Treat blanks as being significant;

NOTE 3.10 (cont.)

(3) Use only the exclamation mark (!) to indicate a comment, but do not start the comment in character position 6;

(4) For continued statements, place an ampersand (&) in both character position 73 of a continued line and character position 6 of a continuing line.

Section 4: Types

Fortran provides an abstract means whereby data may be categorized without relying on a particular physical representation. This abstract means is the concept of type.

An intrinsic type is one that is defined by the language. The intrinsic types are integer, real, complex, character, and logical.

A derived type is one that is defined by a derived-type definition (4.5.1).

A derived type may be used only where its definition is accessible (4.5.1.1). An intrinsic type is always accessible.

A type is specified in several contexts by a **type specifier**.

R401	*type-spec*	**is**	*intrinsic-type-spec*
		or	*derived-type-spec*

C401 (R401) The *derived-type-spec* shall not specify an abstract type (4.5.6).

4.1 The concept of type

A type has a name, a set of valid values, a means to denote such values (constants), and a set of operations to manipulate the values.

NOTE 4.1

For example, the logical type has a set of two values, denoted by the lexical tokens .TRUE. and .FALSE., which are manipulated by logical operations.

An example of a less restricted type is the integer type. This type has a processor-dependent set of integer numeric values, each of which is denoted by an optional sign followed by a string of digits, and which may be manipulated by integer arithmetic operations and relational operations.

4.1.1 Set of values

For each type, there is a set of valid values. The set of valid values may be completely determined, as is the case for logical, or may be determined by a processor-dependent method, as is the case for integer, character, and real. For complex, the set of valid values consists of the set of all the combinations of the values of the individual components. For derived types, the set of valid values is as defined in 4.5.7.

4.1.2 Constants

The syntax for literal constants of each intrinsic type is specified in 4.4.

The syntax for denoting a value indicates the type, type parameters, and the particular value.

A constant value may be given a name (5.1.2.10, 5.2.9).

A structure constructor (4.5.9) may be used to construct a constant value of derived type from an appropriate sequence of initialization expressions (7.1.7). Such a constant value is considered to be a scalar even though the value may have components that are arrays.

4.1.3 Operations

For each of the intrinsic types, a set of operations and corresponding operators is defined intrinsically. These are described in Section 7. The intrinsic set may be augmented with operations and operators defined by functions with the OPERATOR interface (12.3.2.1). Operator definitions are described in Sections 7 and 12.

For derived types, there are no intrinsic operations. Operations on derived types may be defined by the program (4.5.10).

4.2 Type parameters

A type may be parameterized. In this case, the set of values, the syntax for denoting the values, and the set of operations on the values of the type depend on the values of the parameters.

The intrinsic types are all parameterized. Derived types may be defined to be parameterized.

A type parameter is either a kind type parameter or a length type parameter.

A kind type parameter may be used in initialization and specification expressions within the derived-type definition (4.5.1) for the type; it participates in generic resolution (16.2.3). Each of the intrinsic types has a kind type parameter named KIND, which is used to distinguish multiple representations of the intrinsic type.

NOTE 4.2

By design, the value of a kind type parameter is known at compile time. Some parameterizations that involve multiple representation forms need to be distinguished at compile time for practical implementation and performance. Examples include the multiple precisions of the intrinsic real type and the possible multiple character sets of the intrinsic character type.

A type parameter of a derived type may be specified to be a kind type parameter in order to allow generic resolution based on the parameter; that is to allow a single generic to include two specific procedures that have interfaces distinguished only by the value of a kind type parameter of a dummy argument. Generics are designed to be resolvable at compile time.

A length type parameter may be used in specification expressions within the derived-type definition for the type, but it shall not be used in initialization expressions. The intrinsic character type has a length type parameter named LEN, which is the length of the string.

NOTE 4.3

The adjective "length" is used for type parameters other than kind type parameters because they often specify a length, as for intrinsic character type. However, they may be used for other purposes. The important difference from kind type parameters is that their values need not be known at compile time and might change during execution.

A type parameter value may be specified with a type specification (4.4, 4.5.8).

R402	*type-param-value*	**is**	*scalar-int-expr*
		or	*
		or	:

C402 (R402) The *type-param-value* for a kind type parameter shall be an initialization expression.

C403 (R402) A colon may be used as a *type-param-value* only in the declaration of an entity or component that has the POINTER or ALLOCATABLE attribute.

A **deferred type parameter** is a length type parameter whose value can change during execution of the program. A colon as a *type-param-value* specifies a deferred type parameter.

The values of the deferred type parameters of an object are determined by successful execution of an ALLOCATE statement (6.3.1), execution of an intrinsic assignment statement (7.4.1.3), execution of a pointer assignment statement (7.4.2), or by argument association (12.4.1.2).

NOTE 4.4

Deferred type parameters of functions, including function procedure pointers, have no values. Instead, they indicate that those type parameters of the function result will be determined by execution of the function, if it returns an allocated allocatable result or an associated pointer result.

An **assumed type parameter** is a length type parameter for a dummy argument that assumes the type parameter value from the corresponding actual argument; it is also used for an associate name in a SELECT TYPE construct that assumes the type parameter value from the corresponding selector. An asterisk as a *type-param-value* specifies an assumed type parameter.

4.3 Relationship of types and values to objects

The name of a type serves as a type specifier and may be used to declare objects of that type. A declaration specifies the type of a named object. A data object may be declared explicitly or implicitly. Data objects may have attributes in addition to their types. Section 5 describes the way in which a data object is declared and how its type and other attributes are specified.

Scalar data of any intrinsic or derived type may be shaped in a rectangular pattern to compose an array of the same type and type parameters. An array object has a type and type parameters just as a scalar object does.

A variable is a data object. The type and type parameters of a variable determine which values that variable may take. Assignment provides one means of defining or redefining the value of a variable of any type. Assignment is defined intrinsically for all types where the type, type parameters, and shape of both the variable and the value to be assigned to it are identical. Assignment between objects of certain differing intrinsic types, type parameters, and shapes is described in Section 7. A subroutine and a generic interface (4.5.1, 12.3.2.1) whose generic specifier is ASSIGNMENT (=) define an assignment that is not defined intrinsically or redefine an intrinsic derived-type assignment (7.4.1.4).

NOTE 4.5

For example, assignment of a real value to an integer variable is defined intrinsically.

The type of a variable determines the operations that may be used to manipulate the variable.

4.4 Intrinsic types

The intrinsic types are:

numeric types:	integer, real, and complex
nonnumeric types:	character and logical

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

R403	*intrinsic-type-spec*	**is**	INTEGER [*kind-selector*]
		or	REAL [*kind-selector*]
		or	DOUBLE PRECISION
		or	COMPLEX [*kind-selector*]
		or	CHARACTER [*char-selector*]
		or	LOGICAL [*kind-selector*]
R404	*kind-selector*	**is**	([KIND =] *scalar-int-initialization-expr*)

C404 (R404) The value of *scalar-int-initialization-expr* shall be nonnegative and shall specify a representation method that exists on the processor.

4.4.1 Integer type

The set of values for the **integer type** is a subset of the mathematical integers. A processor shall provide one or more **representation methods** that define sets of values for data of type integer. Each such method is characterized by a value for a type parameter called the **kind** type parameter. The kind type parameter of a representation method is returned by the intrinsic inquiry function KIND (13.7.59). The decimal exponent range of a representation method is returned by the intrinsic function RANGE (13.7.96). The intrinsic function SELECTED_INT_KIND (13.7.105) returns a kind value based on a specified decimal range requirement. The integer type includes a zero value, which is considered neither negative nor positive. The value of a signed integer zero is the same as the value of an unsigned integer zero.

The type specifier for the integer type uses the keyword INTEGER.

If the kind type parameter is not specified, the default kind value is KIND (0) and the type specified is **default integer**.

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

R405	*signed-int-literal-constant*	**is**	[*sign*] *int-literal-constant*
R406	*int-literal-constant*	**is**	*digit-string* [_ *kind-param*]
R407	*kind-param*	**is**	*digit-string*
		or	*scalar-int-constant-name*
R408	*signed-digit-string*	**is**	[*sign*] *digit-string*
R409	*digit-string*	**is**	*digit* [*digit*] ...
R410	*sign*	**is**	+
		or	–

C405 (R407) A *scalar-int-constant-name* shall be a named constant of type integer.

C406 (R407) The value of *kind-param* shall be nonnegative.

C407 (R406) The value of *kind-param* shall specify a representation method that exists on the processor.

The optional kind type parameter following *digit-string* specifies the kind type parameter of the integer constant; if it is not present, the constant is of type default integer.

An integer constant is interpreted as a decimal value.

NOTE 4.6

Examples of signed integer literal constants are:

NOTE 4.6 (cont.)

```
473
+56
-101
21_2
21_SHORT
1976354279568241_8
```

where SHORT is a scalar integer named constant.

R411 *boz-literal-constant* **is** *binary-constant*
or *octal-constant*
or *hex-constant*
R412 *binary-constant* **is** B ' *digit* [*digit*] ... '
or B " *digit* [*digit*] ... "

C408 (R412) *digit* shall have one of the values 0 or 1.

R413 *octal-constant* **is** O ' *digit* [*digit*] ... '
or O " *digit* [*digit*] ... "

C409 (R413) *digit* shall have one of the values 0 through 7.

R414 *hex-constant* **is** Z ' *hex-digit* [*hex-digit*] ... '
or Z " *hex-digit* [*hex-digit*] ... "
R415 *hex-digit* **is** *digit*
or A
or B
or C
or D
or E
or F

Binary, octal and hexadecimal constants are interpreted according to their respective number systems. The *hex-digit*s A through F represent the numbers ten through fifteen, respectively; they may be represented by their lower-case equivalents.

C410 (R411) A *boz-literal-constant* shall appear only as a *data-stmt-constant* in a DATA statement, as the actual argument associated with the dummy argument A of the numeric intrinsic functions DBLE, REAL or INT, or as the actual argument associated with the X or Y dummy argument of the intrinsic function CMPLX.

4.4.2 Real type

The **real type** has values that approximate the mathematical real numbers. A processor shall provide two or more **approximation methods** that define sets of values for data of type real. Each such method has a **representation method** and is characterized by a value for a type parameter called the **kind** type parameter. The kind type parameter of an approximation method is returned by the intrinsic inquiry function KIND (13.7.59). The decimal precision and decimal exponent range of an approximation method are returned by the intrinsic functions PRECISION (13.7.90) and RANGE (13.7.96). The intrinsic function SELECTED_REAL_KIND (13.7.106) returns a kind value based on specified precision and decimal range requirements.

NOTE 4.7

> See C.1.2 for remarks concerning selection of approximation methods.

The real type includes a zero value. Processors that distinguish between positive and negative zeros shall treat them as equivalent

(1) in all relational operations,

(2) as actual arguments to intrinsic procedures other than those for which it is explicitly specified that negative zero is distinguished, and

(3) as the *scalar-numeric-expr* in an arithmetic IF.

NOTE 4.8

> On a processor that can distinguish between 0.0 and −0.0,
>
> ```
> ( X >= 0.0 )
> ```
>
> evaluates to true if X = 0.0 or if X = −0.0,
>
> ```
> ( X < 0.0 )
> ```
>
> evaluates to false for X = −0.0, and
>
> ```
> IF (X) 1,2,3
> ```
>
> causes a transfer of control to the branch target statement with the statement label "2" for both X = 0.0 and X = −0.0.
>
> In order to distinguish between 0.0 and −0.0, a program should use the SIGN function. SIGN(1.0,X) will return −1.0 if X < 0.0 or if the processor distinguishes between 0.0 and −0.0 and X has the value −0.0.

The type specifier for the real type uses the keyword REAL. The keyword DOUBLE PRECISION is an alternate specifier for one kind of real type.

If the type keyword REAL is specified and the kind type parameter is not specified, the default kind value is KIND (0.0) and the type specified is **default real**. If the type keyword DOUBLE PRECISION is specified, the kind value is KIND (0.0D0) and the type specified is **double precision real**. The decimal precision of the double precision real approximation method shall be greater than that of the default real method.

R416	*signed-real-literal-constant*	**is**	[*sign*] *real-literal-constant*
R417	*real-literal-constant*	**is**	*significand* [*exponent-letter exponent*] [_ *kind-param*]
		or	*digit-string exponent-letter exponent* [_ *kind-param*]
R418	*significand*	**is**	*digit-string* . [*digit-string*]
		or	. *digit-string*
R419	*exponent-letter*	**is**	E
		or	D
R420	*exponent*	**is**	*signed-digit-string*

C411 (R417) If both *kind-param* and *exponent-letter* are present, *exponent-letter* shall be E.

C412 (R417) The value of *kind-param* shall specify an approximation method that exists on the processor.

A real literal constant without a kind type parameter is a default real constant if it is without an exponent part or has exponent letter E, and is a double precision real constant if it has exponent letter D. A real literal constant written with a kind type parameter is a real constant with the specified kind type parameter.

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

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

NOTE 4.9

Examples of signed real literal constants are:

```
-12.78
+1.6E3
2.1
-16.E4_8
0.45D-4
10.93E7_QUAD
.123
3E4
```

where QUAD is a scalar integer named constant.

4.4.3 Complex type

The **complex type** has values that approximate the mathematical complex numbers. The values of a complex type are ordered pairs of real values. The first real value is called the **real part**, and the second real value is called the **imaginary part**.

Each approximation method used to represent data entities of type real shall be available for both the real and imaginary parts of a data entity of type complex. A **kind** type parameter may be specified for a complex entity and selects for both parts the real approximation method characterized by this kind type parameter value. The kind type parameter of an approximation method is returned by the intrinsic inquiry function KIND (13.7.59).

The type specifier for the complex type uses the keyword COMPLEX. There is no keyword for double precision complex. If the type keyword COMPLEX is specified and the kind type parameter is not specified, the default kind value is the same as that for default real, the type of both parts is default real, and the type specified is **default complex**.

R421	*complex-literal-constant*	**is**	(*real-part* , *imag-part*)
R422	*real-part*	**is**	*signed-int-literal-constant*
		or	*signed-real-literal-constant*
		or	*named-constant*
R423	*imag-part*	**is**	*signed-int-literal-constant*
		or	*signed-real-literal-constant*
		or	*named-constant*

C413 (R421) Each named constant in a complex literal constant shall be of type integer or real.

If the real part and the imaginary part of a complex literal constant are both real, the kind type parameter value of the complex literal constant is the kind type parameter value of the part with the greater decimal precision; if the precisions are the same, it is the kind type parameter value of one of the parts as determined by the processor. If a part has a kind type parameter value different from that of the complex literal constant, the part is converted to the approximation method of the complex literal constant.

If both the real and imaginary parts are integer, they are converted to the default real approximation method and the constant is of type default complex. If only one of the parts is an integer, it is converted to the approximation method selected for the part that is real and the kind type parameter value of the complex literal constant is that of the part that is real.

NOTE 4.10

Examples of complex literal constants are:

```
(1.0, -1.0)
(3, 3.1E6)
(4.0_4, 3.6E7_8)
( 0., PI)
```

where PI is a previously declared named real constant.

4.4.4 Character type

The **character type** has 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; its value is greater than or equal to zero. Strings of different lengths are all of type character.

A processor shall provide one or more **representation methods** that define sets of values for data of type character. Each such method is characterized by a value for a type parameter called the **kind** type parameter. The kind type parameter of a representation method is returned by the intrinsic inquiry function KIND (13.7.59). The intrinsic function SELECTED_CHAR_KIND (13.7.104) returns a kind value based on the name of a character type. Any character of a particular representation method representable in the processor may occur in a character string of that representation method.

The character set defined by ISO/IEC 646:1991 (International Reference Version) is referred to as the **ASCII character set** or the **ASCII character type**. The character set defined by ISO/IEC 10646-1:2000 UCS-4 is referred to as the **ISO 10646 character set** or the **ISO 10646 character type**.

4.4.4.1 Character type specifier

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

If the kind type parameter is not specified, the default kind value is KIND ('A') and the type specified is **default character**.

R424	*char-selector*	**is**	*length-selector*
		or	(LEN = *type-param-value* , ■
			■ KIND = *scalar-int-initialization-expr*)
		or	(*type-param-value* , ■
			■ [KIND =] *scalar-int-initialization-expr*)
		or	(KIND = *scalar-int-initialization-expr* ■
			■ [, LEN =*type-param-value*])
R425	*length-selector*	**is**	([LEN =] *type-param-value*)
		or	* *char-length* [,]
R426	*char-length*	**is**	(*type-param-value*)

or *scalar-int-literal-constant*

C414 (R424) The value of *scalar-int-initialization-expr* shall be nonnegative and shall specify a representation method that exists on the processor.

C415 (R426) The *scalar-int-literal-constant* shall not include a *kind-param*.

C416 (R424 R425 R426) A *type-param-value* of * may be used only in the following ways:

(1) to declare a dummy argument,
(2) to declare a named constant,
(3) in the *type-spec* of an ALLOCATE statement wherein each *allocate-object* is a dummy argument of type CHARACTER with an assumed character length, or
(4) in an external function, to declare the character length parameter of the function result.

C417 A function name shall not be declared with an asterisk *type-param-value* unless it is of type CHARACTER and is the name of the result of an external function or the name of a dummy function.

C418 A function name declared with an asterisk *type-param-value* shall not be an array, a pointer, recursive, or pure.

C419 (R425) The optional comma in a *length-selector* is permitted only in a *declaration-type-spec* in a *type-declaration-stmt*.

C420 (R425) The optional comma in a *length-selector* is permitted only if no double-colon separator appears in the *type-declaration-stmt*.

C421 (R424) The length specified for a character statement function or for a statement function dummy argument of type character shall be an initialization expression.

The *char-selector* in a CHARACTER *intrinsic-type-spec* and the * *char-length* in an *entity-decl* or in a *component-decl* of a type definition specify character length. The * *char-length* in an *entity-decl* or a *component-decl* specifies an individual length and overrides the length specified in the *char-selector*, if any. If a * *char-length* is not specified in an *entity-decl* or a *component-decl*, the *length-selector* or *type-param-value* specified in the *char-selector* is the character length. If the length is not specified in a *char-selector* or a * *char-length*, the length is 1.

If the character length parameter value evaluates to a negative value, the length of character entities declared is zero. A character length parameter value of : indicates a deferred type parameter (4.2). A *char-length* type parameter value of * has the following meaning:

(1) If used to declare a dummy argument of a procedure, the dummy argument assumes the length of the associated actual argument.
(2) If used to declare a named constant, the length is that of the constant value.
(3) If used in the *type-spec* of an ALLOCATE statement, each *allocate-object* assumes its length from the associated actual argument.
(4) If used to specify the character length parameter of a function result, any scoping unit invoking the function shall declare the function name with a character length parameter value other than * or access such a definition by host or use association. When the function is invoked, the length of the result variable in the function is assumed from the value of this type parameter.

4.4.4.2 Character literal constant

A **character literal constant** is written as a sequence of characters, delimited by either apostrophes or quotation marks.

R427	*char-literal-constant*	**is**	[*kind-param* _] ' [*rep-char*] ... '
		or	[*kind-param* _] " [*rep-char*] ... "

C422 (R427) The value of *kind-param* shall specify a representation method that exists on the processor.

The optional kind type parameter preceding the leading delimiter specifies the kind type parameter of the character constant; if it is not present, the constant is of type default character.

For the type character with kind *kind-param*, if present, and for type default character otherwise, a **representable character**, *rep-char*, is defined as follows:

(1) In free source form, it is any graphic character in the processor-dependent character set.

(2) In fixed source form, it is any character in the processor-dependent character set. A processor may restrict the occurrence of some or all of the control characters.

NOTE 4.11

FORTRAN 77 allowed any character to occur in a character context. This standard allows a source program to contain characters of more than one kind. Some processors may identify characters of nondefault kinds by control characters (called "escape" or "shift" characters). It is difficult, if not impossible, to process, edit, and print files where some instances of control characters have their intended meaning and some instances might not. Almost all control characters have uses or effects that effectively preclude their use in character contexts and this is why free source form allows only graphic characters as representable characters. Nevertheless, for compatibility with FORTRAN 77, control characters remain permitted in principle in fixed source form.

The delimiting apostrophes or quotation marks are not part of the value of the character literal constant.

An apostrophe character within a character constant delimited by apostrophes is represented by 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 by two consecutive quotation marks (without intervening blanks) and the two quotation marks are counted as one character.

A zero-length character literal constant is represented by two consecutive apostrophes (without intervening blanks) or two consecutive quotation marks (without intervening blanks) outside of a character context.

The intrinsic operation **concatenation** (//) is defined between two data entities of type character (7.2.2) with the same kind type parameter.

NOTE 4.12

Examples of character literal constants are:

```
"DON'T"
'DON''T'
```

both of which have the value DON'T and

```
''
```

which has the zero-length character string as its value.

NOTE 4.13

An example of a nondefault character literal constant, where the processor supports the corresponding character set, is:

NIHONGO_'彼女なしでは何もできない。'

where NIHONGO is a named constant whose value is the kind type parameter for Nihongo (Japanese) characters.

4.4.4.3 Collating sequence

Each implementation defines a collating sequence for the character set of each kind of character. 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 (13.7.19) and ICHAR (13.7.50) provide conversions between the characters and the integers according to this mapping.

NOTE 4.14

For example:

```
ICHAR ( 'X' )
```

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

For the default character type, the only constraints on the collating sequence are the following:

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

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

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

(4) ICHAR ('a') < ICHAR ('b') < ... < ICHAR ('z') for the twenty-six lower-case letters.

(5) ICHAR (' ') < ICHAR ('0') < ICHAR ('9') < ICHAR ('a') or
ICHAR (' ') < ICHAR ('a') < ICHAR ('z') < ICHAR ('0').

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.

The collating sequence for the ASCII character type is as defined by ISO/IEC 646:1991 (International Reference Version); this collating sequence is called the **ASCII collating sequence** in this standard. The collating sequence for the ISO 10646 character type is as defined by ISO/IEC 10646-1:2000.

NOTE 4.15

The intrinsic functions ACHAR (13.7.2) and IACHAR (13.7.45) provide conversion between characters and corresponding integer values according to the ASCII collating sequence.

The intrinsic functions LGT, LGE, LLE, and LLT (13.7.63-13.7.66) provide comparisons between strings based on the ASCII collating sequence. International portability is guaranteed if the set of characters used is limited to the letters, digits, underscore, and special characters.

4.4.5 Logical type

The **logical type** has two values, which represent true and false.

A processor shall provide one or more **representation methods** for data of type logical. Each such method is characterized by a value for a type parameter called the **kind** type parameter. The kind type parameter of a representation method is returned by the intrinsic inquiry function KIND (13.7.59).

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

If the kind type parameter is not specified, the default kind value is KIND (.FALSE.) and the type specified is **default logical**.

R428	*logical-literal-constant*	**is**	.TRUE. [_ *kind-param*]
		or	.FALSE. [_ *kind-param*]

C423 (R428) The value of *kind-param* shall specify a representation method that exists on the processor.

The optional kind type parameter following the trailing delimiter specifies the kind type parameter of the logical constant; if it is not present, the constant is of type default logical.

The intrinsic operations defined for data entities of logical type are: negation (.NOT.), conjunction (.AND.), inclusive disjunction (.OR.), logical equivalence (.EQV.), and logical nonequivalence (.NEQV.) as described in 7.2.4. There is also a set of intrinsically defined relational operators that compare the values of data entities of other types and yield a value of type default logical. These operations are described in 7.2.3.

4.5 Derived types

Additional types may be derived from the intrinsic types and other derived types. A type definition is required to define the name of the type and the names and attributes of its components and type-bound procedures.

A derived type may be parameterized by multiple type parameters, each of which is defined to be either a kind or length type parameter and may have a default value.

The **ultimate components** of an object of derived type are the components that are of intrinsic type or have the POINTER or ALLOCATABLE attribute, plus the ultimate components of the components of the object that are of derived type and have neither the ALLOCATABLE nor POINTER attribute.

NOTE 4.16

The ultimate components of objects of the derived type `kids` defined below are `name`, `age`, and `other_kids`.

```
type :: person
  character(len=20) :: name
  integer :: age
end type person

type :: kids
  type(person) :: oldest_child
  type(person), allocatable, dimension(:) :: other_kids
end type kids
```

By default, no storage sequence is implied by the order of the component definitions. However, a storage order is implied for a sequence type (4.5.1.2). If the derived type has the BIND attribute, the storage sequence is that required by the companion processor (2.5.10, 15.2.3).

4.5.1 Derived-type definition

R429 *derived-type-def* **is** *derived-type-stmt*
  [*type-param-def-stmt*] ...
  [*private-or-sequence*] ...
  [*component-part*]
  [*type-bound-procedure-part*]
  end-type-stmt

R430 *derived-type-stmt* **is** TYPE [[, *type-attr-spec-list*] ::] *type-name* ■
  ■ [(*type-param-name-list*)]

R431 *type-attr-spec* **is** *access-spec*
  or EXTENDS (*parent-type-name*)
  or ABSTRACT
  or BIND (C)

C424 (R430) A derived type *type-name* shall not be DOUBLEPRECISION or the same as the name of any intrinsic type defined in this standard.

C425 (R430) The same *type-attr-spec* shall not appear more than once in a given *derived-type-stmt*.

C426 (R431) A *parent-type-name* shall be the name of an accessible extensible type (4.5.6).

C427 (R429) If the type definition contains or inherits (4.5.6.1) a deferred binding (4.5.4), ABSTRACT shall appear.

C428 (R429) If ABSTRACT appears, the type shall be extensible.

C429 (R429) If EXTENDS appears, SEQUENCE shall not appear.

R432 *private-or-sequence* **is** *private-components-stmt*
  or *sequence-stmt*

C430 (R429) The same *private-or-sequence* shall not appear more than once in a given *derived-type-def*.

R433 *end-type-stmt* **is** END TYPE [*type-name*]

C431 (R433) If END TYPE is followed by a *type-name*, the *type-name* shall be the same as that in the corresponding *derived-type-stmt*.

Derived types with the BIND attribute are subject to additional constraints as specified in 15.2.3.

NOTE 4.17

An example of a derived-type definition is:

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

An example of declaring a variable CHAIRMAN of type PERSON is:

```
TYPE (PERSON) :: CHAIRMAN
```

4.5.1.1 Accessibility

Types that are defined in a module or accessible in that module by use association have either the PUBLIC or PRIVATE attribute. Types for which an *access-spec* is not explicitly specified in that module have the default accessibility attribute for that module. The default accessibility attribute for a module is PUBLIC unless it has been changed by a PRIVATE statement (5.2.1). Only types that have the PUBLIC attribute in that module are available to be accessed from that module by use association.

The accessibility of a type does not affect, and is not affected by, the accessibility of its components and bindings.

If a type definition is private, then the type name, and thus the structure constructor (4.5.9) for the type, are accessible only within the module containing the definition.

NOTE 4.18

An example of a type with a private name is:

```
TYPE, PRIVATE :: AUXILIARY
   LOGICAL :: DIAGNOSTIC
   CHARACTER (LEN = 20) :: MESSAGE
END TYPE AUXILIARY
```

Such a type would be accessible only within the module in which it is defined.

4.5.1.2 Sequence type

R434 *sequence-stmt* **is** SEQUENCE

C432 (R438) If SEQUENCE appears, all derived types specified in component definitions shall be sequence types.

C433 (R429) If SEQUENCE appears, a *type-bound-procedure-part* shall not appear.

If the **SEQUENCE statement** appears, the type is a **sequence type**. The order of the component definitions in a sequence type specifies a storage sequence for objects of that type. If there are no type parameters and all of the ultimate components of objects of the type are of type default integer, default real, double precision real, default complex, or default logical and are not pointers or allocatable, the type is a **numeric sequence type**. If there are no type parameters and all of the ultimate components of objects of the type are of type default character and are not pointers or allocatable, the type is a **character sequence type**.

NOTE 4.19

An example of a numeric sequence type is:

```
TYPE NUMERIC_SEQ
   SEQUENCE
   INTEGER :: INT_VAL
   REAL    :: REAL_VAL
   LOGICAL :: LOG_VAL
END TYPE NUMERIC_SEQ
```

NOTE 4.20

A structure resolves into a sequence of components. Unless the structure includes a SEQUENCE statement, the use of this terminology in no way implies that these components are stored in

NOTE 4.20 (cont.)

this, or any other, order. Nor is there any requirement that contiguous storage be used. The sequence merely refers to the fact that in writing the definitions there will necessarily be an order in which the components appear, and this will define a sequence of components. This order is of limited significance because a component of an object of derived type will always be accessed by a component name except in the following contexts: the sequence of expressions in a derived-type value constructor, intrinsic assignment, the data values in namelist input data, and the inclusion of the structure in an input/output list of a formatted data transfer, where it is expanded to this sequence of components. Provided the processor adheres to the defined order in these cases, it is otherwise free to organize the storage of the components for any nonsequence structure in memory as best suited to the particular architecture.

4.5.1.3 Determination of derived types

Derived-type definitions with the same type name may appear in different scoping units, in which case they may be independent and describe different derived types or they may describe the same type.

Two data entities have the same type if they are declared with reference to the same derived-type definition. The definition may be accessed from a module or from a host scoping unit. Data entities in different scoping units also have the same type if they are declared with reference to different derived-type definitions that specify the same type name, all have the SEQUENCE property or all have the BIND attribute, have no components with PRIVATE accessibility, and have type parameters and components that agree in order, name, and attributes. Otherwise, they are of different derived types. A data entity declared using a type with the SEQUENCE property or with the BIND attribute is not of the same type as an entity of a type declared to be PRIVATE or that has any components that are PRIVATE.

NOTE 4.21

An example of declaring two entities with reference to the same derived-type definition is:

```
TYPE POINT
   REAL  X,  Y
END TYPE POINT
TYPE (POINT) :: X1
CALL SUB (X1)
   ...
CONTAINS
   SUBROUTINE SUB (A)
      TYPE (POINT) :: A
         ...
   END SUBROUTINE SUB
```

The definition of derived type POINT is known in subroutine SUB by host association. Because the declarations of X1 and A both reference the same derived-type definition, X1 and A have the same type. X1 and A also would have the same type if the derived-type definition were in a module and both SUB and its containing program unit accessed the module.

NOTE 4.22

An example of data entities in different scoping units having the same type is:

```
PROGRAM PGM
   TYPE EMPLOYEE
      SEQUENCE
```

NOTE 4.22 (cont.)

```
      INTEGER        ID_NUMBER
      CHARACTER (50) NAME
   END TYPE EMPLOYEE
   TYPE (EMPLOYEE) PROGRAMMER
   CALL SUB (PROGRAMMER)
      ...
END PROGRAM PGM
SUBROUTINE SUB (POSITION)
   TYPE EMPLOYEE
      SEQUENCE
      INTEGER        ID_NUMBER
      CHARACTER (50) NAME
   END TYPE EMPLOYEE
   TYPE (EMPLOYEE) POSITION
   ...
END SUBROUTINE SUB
```

The actual argument PROGRAMMER and the dummy argument POSITION have the same type because they are declared with reference to a derived-type definition with the same name, the SEQUENCE property, and components that agree in order, name, and attributes.

Suppose the component name ID_NUMBER was ID_NUM in the subroutine. Because all the component names are not identical to the component names in derived type EMPLOYEE in the main program, the actual argument PROGRAMMER would not be of the same type as the dummy argument POSITION. Thus, the program would not be standard conforming.

NOTE 4.23

The requirement that the two types have the same name applies to the *type-name*s of the respective *derived-type-stmt*s, not to local names introduced via renaming in USE statements.

4.5.2 Derived-type parameters

R435 *type-param-def-stmt* **is** INTEGER [*kind-selector*] , *type-param-attr-spec* :: ■
■ *type-param-decl-list*

R436 *type-param-decl* **is** *type-param-name* [= *scalar-int-initialization-expr*]

C434 (R435) A *type-param-name* in a *type-param-def-stmt* in a *derived-type-def* shall be one of the *type-param-name*s in the *derived-type-stmt* of that *derived-type-def*.

C435 (R435) Each *type-param-name* in the *derived-type-stmt* in a *derived-type-def* shall appear as a *type-param-name* in a *type-param-def-stmt* in that *derived-type-def*.

R437 *type-param-attr-spec* **is** KIND
or LEN

The derived type is parameterized if the *derived-type-stmt* has any *type-param-name*s.

Each type parameter is itself of type integer. If its kind selector is omitted, the kind type parameter is default integer.

The *type-param-attr-spec* explicitly specifies whether a type parameter is a kind parameter or a length parameter.

If a *type-param-decl* has a *scalar-int-initialization-expr*, the type parameter has a default value which is specified by the expression. If necessary, the value is converted according to the rules of intrinsic assignment (7.4.1.3) to a value of the same kind as the type parameter.

A type parameter may be used as a primary in a specification expression (7.1.6) in the *derived-type-def*. A kind type parameter may also be used as a primary in an initialization expression (7.1.7) in the *derived-type-def*.

NOTE 4.24

The following example uses derived-type parameters.

```
TYPE humongous_matrix(k, d)
  INTEGER, KIND :: k = kind(0.0)
  INTEGER(selected_int_kind(12)), LEN :: d
    !-- Specify a nondefault kind for d.
  REAL(k) :: element(d,d)
END TYPE
```

In the following example, `dim` is declared to be a kind parameter, allowing generic overloading of procedures distinguished only by `dim`.

```
TYPE general_point(dim)
  INTEGER, KIND :: dim
  REAL :: coordinates(dim)
END TYPE
```

4.5.2.1 Type parameter order

Type parameter order is an ordering of the type parameters of a derived type; it is used for derived-type specifiers.

The type parameter order of a nonextended type is the order of the type parameter list in the derived-type definition. The type parameter order of an extended type consists of the type parameter order of its parent type followed by any additional type parameters in the order of the type parameter list in the derived-type definition.

NOTE 4.25

Given

```
TYPE :: t1(k1,k2)
  INTEGER,KIND :: k1,k2
  REAL(k1) a(k2)
END TYPE
TYPE,EXTENDS(t1) :: t2(k3)
  INTEGER,KIND :: k3
  LOGICAL(k3) flag
END TYPE
```

the type parameter order for type T1 is K1 then K2, and the type parameter order for type T2 is K1 then K2 then K3.

4.5.3 Components

R438 *component-part* **is** [*component-def-stmt*] ...

R439	*component-def-stmt*	**is**	*data-component-def-stmt*
		or	*proc-component-def-stmt*
R440	*data-component-def-stmt*	**is**	*declaration-type-spec* [[, *component-attr-spec-list*] ::] ■
			■ *component-decl-list*
R441	*component-attr-spec*	**is**	POINTER
		or	DIMENSION (*component-array-spec*)
		or	ALLOCATABLE
		or	*access-spec*
R442	*component-decl*	**is**	*component-name* [(*component-array-spec*)] ■
			■ [* *char-length*] [*component-initialization*]
R443	*component-array-spec*	**is**	*explicit-shape-spec-list*
		or	*deferred-shape-spec-list*
R444	*component-initialization*	**is**	= *initialization-expr*
		or	=> *null-init*

C436 (R440) No *component-attr-spec* shall appear more than once in a given *component-def-stmt*.

C437 (R440) A component declared with the CLASS keyword (5.1.1.2) shall have the ALLOCATABLE or POINTER attribute.

C438 (R440) If the POINTER attribute is not specified for a component, the *declaration-type-spec* in the *component-def-stmt* shall specify an intrinsic type or a previously defined derived type.

C439 (R440) If the POINTER attribute is specified for a component, the *declaration-type-spec* in the *component-def-stmt* shall specify an intrinsic type or any accessible derived type including the type being defined.

C440 (R440) If the POINTER or ALLOCATABLE attribute is specified, each *component-array-spec* shall be a *deferred-shape-spec-list*.

C441 (R440) If neither the POINTER attribute nor the ALLOCATABLE attribute is specified, each *component-array-spec* shall be an *explicit-shape-spec-list*.

C442 (R443) Each bound in the *explicit-shape-spec* shall either be an initialization expression or be a specification expression that does not contain references to specification functions or any object designators other than named constants or subobjects thereof.

C443 (R440) A component shall not have both the ALLOCATABLE and the POINTER attribute.

C444 (R442) The * *char-length* option is permitted only if the type specified is character.

C445 (R439) Each *type-param-value* within a *component-def-stmt* shall either be a colon, be an initialization expression, or be a specification expression that contains neither references to specification functions nor any object designators other than named constants or subobjects thereof.

NOTE 4.26

Because a type parameter is not an object, a *type-param-value* or a bound in an *explicit-shape-spec* may contain a *type-param-name*.

C446 (R440) If *component-initialization* appears, a double-colon separator shall appear before the *component-decl-list*.

C447 (R440) If => appears in *component-initialization*, POINTER shall appear in the *component-attr-spec-list*. If = appears in *component-initialization*, POINTER or ALLOCATABLE shall not appear in the *component-attr-spec-list*.

R445 *proc-component-def-stmt* **is** PROCEDURE ([*proc-interface*]) , ■
■ *proc-component-attr-spec-list* :: *proc-decl-list*

NOTE 4.27

See 12.3.2.3 for definitions of *proc-interface* and *proc-decl*.

R446 *proc-component-attr-spec* **is** POINTER
or PASS [(*arg-name*)]
or NOPASS
or *access-spec*

C448 (R445) The same *proc-component-attr-spec* shall not appear more than once in a given *proc-component-def-stmt*.

C449 (R445) POINTER shall appear in each *proc-component-attr-spec-list*.

C450 (R445) If the procedure pointer component has an implicit interface or has no arguments, NOPASS shall be specified.

C451 (R445) If PASS (*arg-name*) appears, the interface shall have a dummy argument named *arg-name*.

C452 (R445) PASS and NOPASS shall not both appear in the same *proc-component-attr-spec-list*.

4.5.3.1 Array components

A data component is an array if its *component-decl* contains a *component-array-spec* or its *data-component-def-stmt* contains the DIMENSION attribute. If the *component-decl* contains a *component-array-spec*, it specifies the array rank, and if the array is explicit shape (5.1.2.5.1), the array bounds; otherwise, the *component-array-spec* in the DIMENSION attribute specifies the array rank, and if the array is explicit shape, the array bounds.

NOTE 4.28

An example of a derived type definition with an array component is:

```
TYPE LINE
   REAL, DIMENSION (2, 2) :: COORD    !
                                      ! COORD(:,1) has the value of (/X1, Y1/)
                                      ! COORD(:,2) has the value of (/X2, Y2/)
   REAL                   :: WIDTH    ! Line width in centimeters
   INTEGER                :: PATTERN  ! 1 for solid, 2 for dash, 3 for dot
END TYPE LINE
```

An example of declaring a variable LINE_SEGMENT to be of the type LINE is:

```
TYPE (LINE)         :: LINE_SEGMENT
```

The scalar variable LINE_SEGMENT has a component that is an array. In this case, the array is a subobject of a scalar. The double colon in the definition for COORD is required; the double colon in the definition for WIDTH and PATTERN is optional.

NOTE 4.29

An example of a derived type definition with an allocatable component is:

NOTE 4.29 (cont.)

```
TYPE STACK
   INTEGER              :: INDEX
   INTEGER, ALLOCATABLE :: CONTENTS (:)
END TYPE STACK
```

For each scalar variable of type STACK, the shape of the component CONTENTS is determined by execution of an ALLOCATE statement or assignment statement, or by argument association.

NOTE 4.30

Default initialization of an explicit-shape array component may be specified by an initialization expression consisting of an array constructor (4.7), or of a single scalar that becomes the value of each array element.

4.5.3.2 Pointer components

A component is a pointer (2.4.6) if its *component-attr-spec-list* contains the POINTER attribute. A pointer component may be a data pointer or a procedure pointer.

NOTE 4.31

An example of a derived type definition with a pointer component is:

```
TYPE REFERENCE
   INTEGER                            :: VOLUME, YEAR, PAGE
   CHARACTER (LEN = 50)               :: TITLE
   CHARACTER, DIMENSION (:), POINTER :: SYNOPSIS
END TYPE REFERENCE
```

Any object of type REFERENCE will have the four nonpointer components VOLUME, YEAR, PAGE, and TITLE, plus a pointer to an array of characters holding SYNOPSIS. The size of this target array will be determined by the length of the abstract. The space for the target may be allocated (6.3.1) or the pointer component may be associated with a target by a pointer assignment statement (7.4.2).

4.5.3.3 The passed-object dummy argument

A **passed-object dummy argument** is a distinguished dummy argument of a procedure pointer component or type-bound procedure. It affects procedure overriding (4.5.6.2) and argument association (12.4.1.1).

If NOPASS is specified, the procedure pointer component or type-bound procedure has no passed-object dummy argument.

If neither PASS nor NOPASS is specified or PASS is specified without *arg-name*, the first dummy argument of a procedure pointer component or type-bound procedure is its passed-object dummy argument.

If PASS (*arg-name*) is specified, the dummy argument named *arg-name* is the passed-object dummy argument of the procedure pointer component or named type-bound procedure.

C453 The passed-object dummy argument shall be a scalar, nonpointer, nonallocatable dummy data object with the same declared type as the type being defined; all of its length type parameters shall be assumed; it shall be polymorphic (5.1.1.2) if and only if the type being defined is extensible (4.5.6).

NOTE 4.32

If a procedure is bound to several types as a type-bound procedure, different dummy arguments might be the passed-object dummy argument in different contexts.

4.5.3.4 Default initialization for components

Default initialization provides a means of automatically initializing pointer components to be disassociated, and nonpointer nonallocatable components to have a particular value. Allocatable components are always initialized to not allocated.

If *null-init* appears for a pointer component, that component in any object of the type has an initial association status of disassociated (16.4.2.1) or becomes disassociated as specified in 16.4.2.1.2.

If *initialization-expr* appears for a nonpointer component, that component in any object of the type is initially defined (16.5.3) or becomes defined as specified in 16.5.5 with the value determined from *initialization-expr*. If necessary, the value is converted according to the rules of intrinsic assignment (7.4.1.3) to a value that agrees in type, type parameters, and shape with the component. If the component is of a type for which default initialization is specified for a component, the default initialization specified by *initialization-expr* overrides the default initialization specified for that component. When one initialization **overrides** another it is as if only the overriding initialization were specified (see Note 4.34). Explicit initialization in a type declaration statement (5.1) overrides default initialization (see Note 4.33). Unlike explicit initialization, default initialization does not imply that the object has the SAVE attribute.

A subcomponent (6.1.2) is **default-initialized** if the type of the object of which it is a component specifies default initialization for that component, and the subcomponent is not a subobject of an object that is default-initialized or explicitly initialized.

NOTE 4.33

It is not required that initialization be specified for each component of a derived type. For example:

```
TYPE DATE
    INTEGER DAY
    CHARACTER (LEN = 5) MONTH
    INTEGER :: YEAR = 1994       ! Partial default initialization
END TYPE DATE
```

In the following example, the default initial value for the YEAR component of TODAY is overridden by explicit initialization in the type declaration statement:

```
TYPE (DATE), PARAMETER :: TODAY = DATE (21, "Feb.", 1995)
```

NOTE 4.34

The default initial value of a component of derived type may be overridden by default initialization specified in the definition of the type. Continuing the example of Note 4.33:

```
TYPE SINGLE_SCORE
    TYPE(DATE) :: PLAY_DAY = TODAY
    INTEGER SCORE
    TYPE(SINGLE_SCORE), POINTER :: NEXT => NULL ( )
```

NOTE 4.34 (cont.)

```
END TYPE SINGLE_SCORE
TYPE(SINGLE_SCORE) SETUP
```

The PLAY_DAY component of SETUP receives its initial value from TODAY, overriding the initialization for the YEAR component.

NOTE 4.35

Arrays of structures may be declared with elements that are partially or totally initialized by default. Continuing the example of Note 4.34 :

```
TYPE MEMBER (NAME_LEN)
    INTEGER, LEN :: NAME_LEN
    CHARACTER (LEN = NAME_LEN) NAME = ''
    INTEGER :: TEAM_NO, HANDICAP = 0
    TYPE (SINGLE_SCORE), POINTER :: HISTORY => NULL ( )
END TYPE MEMBER
TYPE (MEMBER(9)) LEAGUE (36)        ! Array of partially initialized elements
TYPE (MEMBER(9)) :: ORGANIZER = MEMBER ("I. Manage",1,5,NULL ( ))
```

ORGANIZER is explicitly initialized, overriding the default initialization for an object of type MEMBER.

Allocated objects may also be initialized partially or totally. For example:

```
ALLOCATE (ORGANIZER % HISTORY)   ! A partially initialized object of type
                                 ! SINGLE_SCORE is created.
```

NOTE 4.36

A pointer component of a derived type may have as its target an object of that derived type. The type definition may specify that in objects declared to be of this type, such a pointer is default initialized to disassociated. For example:

```
TYPE NODE
   INTEGER              :: VALUE = 0
   TYPE (NODE), POINTER :: NEXT_NODE => NULL ( )
END TYPE
```

A type such as this may be used to construct linked lists of objects of type NODE. See C.1.5 for an example.

4.5.3.5 Component order

Component order is an ordering of the nonparent components of a derived type; it is used for intrinsic formatted input/output and structure constructors (where component keywords are not used). Parent components are excluded from the component order of an extended type (4.5.6).

The component order of a nonextended type is the order of the declarations of the components in the derived-type definition. The component order of an extended type consists of the component order of its parent type followed by any additional components in the order of their declarations in the extended derived-type definition.

NOTE 4.37

Given the same type definitions as in Note 4.5.2.1, the component order of type T1 is just A (there is only one component), and the component order of type T2 is A then FLAG. The parent component (T1) does not participate in the component order.

4.5.3.6 Component accessibility

R447 *private-components-stmt* **is** PRIVATE

C454 (R447) A *private-components-stmt* is permitted only if the type definition is within the specification part of a module.

The default accessibility for the components that are declared in a type's *component-part* is private if the type definition contains a *private-components-stmt*, and public otherwise. The accessibility of a component may be explicitly declared by an *access-spec*; otherwise its accessibility is the default for the type definition in which it is declared.

If a component is private, that component name is accessible only within the module containing the definition.

NOTE 4.38

Type parameters are not components. They are effectively always public.

NOTE 4.39

The accessibility of the components of a type is independent of the accessibility of the type name. It is possible to have all four combinations: a public type name with a public component, a private type name with a private component, a public type name with a private component, and a private type name with a public component.

NOTE 4.40

An example of a type with private components is:

```
MODULE DEFINITIONS
   TYPE POINT
      PRIVATE
      REAL :: X, Y
   END TYPE POINT
END MODULE DEFINITIONS
```

Such a type definition is accessible in any scoping unit accessing the module via a USE statement; however, the components X and Y are accessible only within the module.

NOTE 4.41

The following example illustrates the use of an individual component *access-spec* to override the default accessibility:

```
     TYPE MIXED
       PRIVATE
       INTEGER :: I
       INTEGER, PUBLIC :: J
     END TYPE MIXED
```

NOTE 4.41 (cont.)

```
    TYPE (MIXED) :: M
```

The component M%J is accessible in any scoping unit where M is accessible; M%I is accessible only within the module containing the TYPE MIXED definition.

4.5.4 Type-bound procedures

R448	*type-bound-procedure-part*	**is**	*contains-stmt*
			[*binding-private-stmt*]
			proc-binding-stmt
			[*proc-binding-stmt*] ...
R449	*binding-private-stmt*	**is**	PRIVATE

C455 (R448) A *binding-private-stmt* is permitted only if the type definition is within the specification part of a module.

R450	*proc-binding-stmt*	**is**	*specific-binding*
		or	*generic-binding*
		or	*final-binding*
R451	*specific-binding*	**is**	PROCEDURE [(*interface-name*)] ■
			■ [[, *binding-attr-list*] ::] ■
			■ *binding-name* [=> *procedure-name*]

C456 (R451) If => *procedure-name* appears, the double-colon separator shall appear.

C457 (R451) If => *procedure-name* appears, *interface-name* shall not appear.

C458 (R451) The *procedure-name* shall be the name of an accessible module procedure or an external procedure that has an explicit interface.

If neither => *procedure-name* nor *interface-name* appears, it is as though => *procedure-name* had appeared with a procedure name the same as the binding name.

R452	*generic-binding*	**is**	GENERIC ■
			■ [, *access-spec*] :: *generic-spec* => *binding-name-list*

C459 (R452) Within the *specification-part* of a module, each *generic-binding* shall specify, either implicitly or explicitly, the same accessibility as every other *generic-binding* with that *generic-spec* in the same derived type.

C460 (R452) Each *binding-name* in *binding-name-list* shall be the name of a specific binding of the type.

C461 (R452) If *generic-spec* is not *generic-name*, each of its specific bindings shall have a passed-object dummy argument (4.5.3.3).

C462 (R452) If *generic-spec* is OPERATOR (*defined-operator*), the interface of each binding shall be as specified in 12.3.2.1.1.

C463 (R452) If *generic-spec* is ASSIGNMENT (=), the interface of each binding shall be as specified in 12.3.2.1.2.

C464 (R452) If *generic-spec* is *dtio-generic-spec*, the interface of each binding shall be as specified in 9.5.3.7. The type of the `dtv` argument shall be *type-name*.

R453	*binding-attr*	**is**	PASS [(*arg-name*)]
		or	NOPASS
		or	NON_OVERRIDABLE
		or	DEFERRED
		or	*access-spec*

C465 (R453) The same *binding-attr* shall not appear more than once in a given *binding-attr-list*.

C466 (R451) If the interface of the binding has no dummy argument of the type being defined, NOPASS shall appear.

C467 (R451) If PASS (*arg-name*) appears, the interface of the binding shall have a dummy argument named *arg-name*.

C468 (R453) PASS and NOPASS shall not both appear in the same *binding-attr-list*.

C469 (R453) NON_OVERRIDABLE and DEFERRED shall not both appear in the same *binding-attr-list*.

C470 (R453) DEFERRED shall appear if and only if *interface-name* appears.

C471 (R451) An overriding binding (4.5.6.2) shall have the DEFERRED attribute only if the binding it overrides is deferred.

C472 (R451) A binding shall not override an inherited binding (4.5.6.1) that has the NON_OVERRIDABLE attribute.

Each binding in a *proc-binding-stmt* specifies a **type-bound procedure**. A type-bound procedure may have a passed-object dummy argument (4.5.3.3). A *generic-binding* specifies a type-bound generic interface for its specific bindings. A binding that specifies the DEFERRED attribute is a **deferred binding**. A deferred binding shall appear only in the definition of an abstract type.

A type-bound procedure may be identified by a **binding name** in the scope of the type definition. This name is the *binding-name* for a specific binding, and the *generic-name* for a generic binding whose *generic-spec* is *generic-name*. A final binding, or a generic binding whose *generic-spec* is not *generic-name*, has no binding name.

The interface of a specific binding is that of the procedure specified by *procedure-name* or the interface specified by *interface-name*.

NOTE 4.42

An example of a type and a type-bound procedure is:

```
TYPE POINT
  REAL :: X, Y
CONTAINS
  PROCEDURE, PASS :: LENGTH => POINT_LENGTH
END TYPE POINT
...
```

and in the *module-subprogram-part* of the same module:

```
REAL FUNCTION POINT_LENGTH (A, B)
  CLASS (POINT), INTENT (IN) :: A, B
  POINT_LENGTH = SQRT ( (A%X - B%X)**2 + (A%Y - B%Y)**2 )
END FUNCTION POINT_LENGTH
```

The same *generic-spec* may be used in several *generic-binding*s within a single derived-type definition. Each additional *generic-binding* with the same *generic-spec* extends the generic interface.

NOTE 4.43

Unlike the situation with generic procedure names, a generic type-bound procedure name is not permitted to be the same as a specific type-bound procedure name in the same type (16.2).

The default accessibility for the procedure bindings of a type is private if the type definition contains a *binding-private-stmt*, and public otherwise. The accessibility of a procedure binding may be explicitly declared by an *access-spec*; otherwise its accessibility is the default for the type definition in which it is declared.

A public type-bound procedure is accessible via any accessible object of the type. A private type-bound procedure is accessible only within the module containing the type definition.

NOTE 4.44

The accessibility of a type-bound procedure is not affected by a PRIVATE statement in the *component-part*; the accessibility of a data component is not affected by a PRIVATE statement in the *type-bound-procedure-part*.

4.5.5 Final subroutines

R454 *final-binding* **is** FINAL [::] *final-subroutine-name-list*

C473 (R454) A *final-subroutine-name* shall be the name of a module procedure with exactly one dummy argument. That argument shall be nonoptional and shall be a nonpointer, nonallocatable, nonpolymorphic variable of the derived type being defined. All length type parameters of the dummy argument shall be assumed. The dummy argument shall not be INTENT(OUT).

C474 (R454) A *final-subroutine-name* shall not be one previously specified as a final subroutine for that type.

C475 (R454) A final subroutine shall not have a dummy argument with the same kind type parameters and rank as the dummy argument of another final subroutine of that type.

The FINAL keyword specifies a list of **final subroutines**. A final subroutine might be executed when a data entity of that type is finalized (4.5.5.1).

A derived type is **finalizable** if it has any final subroutines or if it has any nonpointer, nonallocatable component whose type is finalizable. A nonpointer data entity is finalizable if its type is finalizable.

NOTE 4.45

Final subroutines are effectively always “accessible”. They are called for entity finalization regardless of the accessibility of the type, its other type-bound procedures, or the subroutine name itself.

NOTE 4.46

Final subroutines are not inherited through type extension and cannot be overridden. The final subroutines of the parent type are called after any additional final subroutines of an extended type are called.

4.5.5.1 The finalization process

Only finalizable entities are finalized. When an entity is **finalized**, the following steps are carried out in sequence:

(1) If the dynamic type of the entity has a final subroutine whose dummy argument has the same kind type parameters and rank as the entity being finalized, it is called with the entity as an actual argument. Otherwise, if there is an elemental final subroutine whose dummy argument has the same kind type parameters as the entity being finalized, it is called with the entity as an actual argument. Otherwise, no subroutine is called at this point.

(2) Each finalizable component that appears in the type definition is finalized. If the entity being finalized is an array, each finalizable component of each element of that entity is finalized separately.

(3) If the entity is of extended type and the parent type is finalizable, the parent component is finalized.

If several entities are to be finalized as a consequence of an event specified in 4.5.5.2, the order in which they are finalized is processor-dependent. A final subroutine shall not reference or define an object that has already been finalized.

If an object is not finalized, it retains its definition status and does not become undefined.

4.5.5.2 When finalization occurs

When a pointer is deallocated its target is finalized. When an allocatable entity is deallocated, it is finalized.

A nonpointer, nonallocatable object that is not a dummy argument or function result is finalized immediately before it would become undefined due to execution of a RETURN or END statement (16.5.6, item (3)). If the object is defined in a module and there are no longer any active procedures referencing the module, it is processor-dependent whether it is finalized.

If an executable construct references a function, the result is finalized after execution of the innermost executable construct containing the reference.

If an executable construct references a structure constructor, the entity created by the structure constructor is finalized after execution of the innermost executable construct containing the reference.

If a specification expression in a scoping unit references a function, the result is finalized before execution of the first executable statement in the scoping unit.

When a procedure is invoked, a nonpointer, nonallocatable object that is an actual argument associated with an INTENT(OUT) dummy argument is finalized.

When an intrinsic assignment statement is executed, *variable* is finalized after evaluation of *expr* and before the definition of *variable*.

NOTE 4.47

If finalization is used for storage management, it often needs to be combined with defined assignment.

If an object is allocated via pointer allocation and later becomes unreachable due to all pointers to that object having their pointer association status changed, it is processor dependent whether it is finalized. If it is finalized, it is processor dependent as to when the final subroutines are called.

4.5.5.3 Entities that are not finalized

If program execution is terminated, either by an error (e.g. an allocation failure) or by execution of a STOP or END PROGRAM statement, entities existing immediately prior to termination are not finalized.

NOTE 4.48

A nonpointer, nonallocatable object that has the SAVE attribute or that occurs in the main program is never finalized as a direct consequence of the execution of a RETURN or END statement.

A variable in a module is not finalized if it retains its definition status and value, even when there is no active procedure referencing the module.

4.5.6 Type extension

A nonsequence derived type that does not have the BIND attribute is an **extensible type**.

An extensible type that does not have the EXTENDS attribute is a **base type**. A type that has the EXTENDS attribute is an **extended type**. The **parent type** of an extended type is the type named in the EXTENDS attribute specification.

NOTE 4.49

The name of the parent type might be a local name introduced via renaming in a USE statement.

A base type is an **extension type** of itself only. An extended type is an extension of itself and of all types for which its parent type is an extension.

An **abstract type** is a type that has the ABSTRACT attribute.

NOTE 4.50

A deferred binding (4.5.4) defers the implementation of a type-bound procedure to extensions of the type; it may appear only in an abstract type. The dynamic type of an object cannot be abstract; therefore, a deferred binding cannot be invoked. An extension of an abstract type need not be abstract if it has no deferred bindings. A short example of an abstract type is:

```
TYPE, ABSTRACT :: FILE_HANDLE
CONTAINS
   PROCEDURE(OPEN_FILE), DEFERRED, PASS(HANDLE) :: OPEN
   ...
END TYPE
```

For a more elaborate example see C.1.4.

4.5.6.1 Inheritance

An extended type includes all of the type parameters, all of the components, and the nonoverridden (4.5.6.2) nonfinal procedure bindings of its parent type. These are said to be **inherited** by the extended type from the parent type. They retain all of the attributes that they had in the parent type. Additional type parameters, components, and procedure bindings may be declared in the derived-type definition of the extended type.

NOTE 4.51

Inaccessible components and bindings of the parent type are also inherited, but they remain inaccessible in the extended type. Inaccessible entities occur if the type being extended is accessed via use association and has a private entity.

NOTE 4.52

A base type is not required to have any components, bindings, or parameters; an extended type is not required to have more components, bindings, or parameters than its parent type.

An extended type has a scalar, nonpointer, nonallocatable, **parent component** with the type and type parameters of the parent type. The name of this component is the parent type name. It has the accessibility of the parent type. Components of the parent component are **inheritance associated** (16.4.4) with the corresponding components inherited from the parent type. An **ancestor component** of a type is the parent component of the type or an ancestor component of the parent component.

NOTE 4.53

A component or type parameter declared in an extended type shall not have the same name as any accessible component or type parameter of its parent type.

NOTE 4.54

Examples:

```
TYPE POINT                          ! A base type
  REAL :: X, Y
END TYPE POINT

TYPE, EXTENDS(POINT) :: COLOR_POINT   ! An extension of TYPE(POINT)
  ! Components X and Y, and component name POINT, inherited from parent
  INTEGER :: COLOR
END TYPE COLOR_POINT
```

4.5.6.2 Type-bound procedure overriding

If a nongeneric binding specified in a type definition has the same binding name as a binding from the parent type then the binding specified in the type definition **overrides** the one from the parent type.

The overriding binding and the overridden binding shall satisfy the following conditions:

(1) Either both shall have a passed-object dummy argument or neither shall.
(2) If the overridden binding is pure then the overriding binding shall also be pure.
(3) Either both shall be elemental or neither shall.
(4) They shall have the same number of dummy arguments.
(5) Passed-object dummy arguments, if any, shall correspond by name and position.
(6) Dummy arguments that correspond by position shall have the same names and characteristics, except for the type of the passed-object dummy arguments.
(7) Either both shall be subroutines or both shall be functions having the same result characteristics (12.2.2).
(8) If the overridden binding is PUBLIC then the overriding binding shall not be PRIVATE.

NOTE 4.55

The following is an example of procedure overriding, expanding on the example in Note 4.42.

```
TYPE, EXTENDS (POINT) :: POINT_3D
  REAL :: Z
CONTAINS
  PROCEDURE, PASS :: LENGTH => POINT_3D_LENGTH
END TYPE POINT_3D
...
```

and in the *module-subprogram-part* of the same module:

```
REAL FUNCTION POINT_3D_LENGTH ( A, B )
  CLASS (POINT_3D), INTENT (IN) :: A
  CLASS (POINT), INTENT (IN) :: B
  SELECT TYPE(B)
    CLASS IS(POINT_3D)
      POINT_3D_LENGTH = SQRT( (A%X-B%X)**2 + (A%Y-B%Y)**2 + (A%Z-B%Z)**2 )
      RETURN
  END SELECT
  PRINT *, 'In POINT_3D_LENGTH, dynamic type of argument is incorrect.'
  STOP
END FUNCTION POINT_3D_LENGTH
```

If a generic binding specified in a type definition has the same *generic-spec* as an inherited binding, it extends the generic interface and shall satisfy the requirements specified in 16.2.3.

If a generic binding in a type definition has the same *dtio-generic-spec* as one inherited from the parent, it extends the type-bound generic interface for *dtio-generic-spec* and shall satisfy the requirements specified in 16.2.3.

A binding of a type and a binding of an extension of that type are said to correspond if the latter binding is the same binding as the former, overrides a corresponding binding, or is an inherited corresponding binding.

4.5.7 Derived-type values

The **component value** of

(1) a pointer component is its pointer association;
(2) an allocatable component is its allocation status and, if it is allocated, its dynamic type and type parameters, bounds and value;
(3) a nonpointer nonallocatable component is its value.

The set of values of a particular derived type consists of all possible sequences of the component values of its components.

4.5.8 Derived-type specifier

A derived-type specifier is used in several contexts to specify a particular derived type and type parameters.

R455 *derived-type-spec* **is** *type-name* [(*type-param-spec-list*)]
R456 *type-param-spec* **is** [*keyword* =] *type-param-value*

C476 (R455) *type-name* shall be the name of an accessible derived type.

C477 (R455) *type-param-spec-list* shall appear only if the type is parameterized.

C478 (R455) There shall be at most one *type-param-spec* corresponding to each parameter of the type. If a type parameter does not have a default value, there shall be a *type-param-spec* corresponding to that type parameter.

C479 (R456) The *keyword*= may be omitted from a *type-param-spec* only if the *keyword*= has been omitted from each preceding *type-param-spec* in the *type-param-spec-list*.

C480 (R456) Each *keyword* shall be the name of a parameter of the type.

C481 (R456) An asterisk may be used as a *type-param-value* in a *type-param-spec* only in the declaration of a dummy argument or associate name or in the allocation of a dummy argument.

Type parameter values that do not have type parameter keywords specified correspond to type parameters in type parameter order (4.5.2.1). If a type parameter keyword is present, the value is assigned to the type parameter named by the keyword. If necessary, the value is converted according to the rules of intrinsic assignment (7.4.1.3) to a value of the same kind as the type parameter.

The value of a type parameter for which no *type-param-value* has been specified is its default value.

4.5.9 Construction of derived-type values

A derived-type definition implicitly defines a corresponding **structure constructor** that allows construction of values of that derived type. The type and type parameters of a constructed value are specified by a derived type specifier.

R457 *structure-constructor* **is** *derived-type-spec* ([*component-spec-list*])
R458 *component-spec* **is** [*keyword* =] *component-data-source*
R459 *component-data-source* **is** *expr*
or *data-target*
or *proc-target*

C482 (R457) The *derived-type-spec* shall not specify an abstract type (4.5.6).

C483 (R457) At most one *component-spec* shall be provided for a component.

C484 (R457) If a *component-spec* is provided for a component, no *component-spec* shall be provided for any component with which it is inheritance associated.

C485 (R457) A *component-spec* shall be provided for a component unless it has default initialization or is inheritance associated with another component for which a *component-spec* is provided or that has default initialization.

C486 (R458) The *keyword*= may be omitted from a *component-spec* only if the *keyword*= has been omitted from each preceding *component-spec* in the constructor.

C487 (R458) Each *keyword* shall be the name of a component of the type.

C488 (R457) The type name and all components of the type for which a *component-spec* appears shall be accessible in the scoping unit containing the structure constructor.

C489 (R457) If *derived-type-spec* is a type name that is the same as a generic name, the *component-*

spec-list shall not be a valid *actual-arg-spec-list* for a function reference that is resolvable as a generic reference (12.4.4.1).

C490 (R459) A *data-target* shall correspond to a nonprocedure pointer component; a *proc-target* shall correspond to a procedure pointer component.

C491 (R459) A *data-target* shall have the same rank as its corresponding component.

NOTE 4.56

The form 'name(...)' is interpreted as a generic *function-reference* if possible; it is interpreted as a *structure-constructor* only if it cannot be interpreted as a generic *function-reference*.

In the absence of a component keyword, each *component-data-source* is assigned to the corresponding component in component order (4.5.3.5). If a component keyword appears, the *expr* is assigned to the component named by the keyword. For a nonpointer component, the declared type and type parameters of the component and *expr* shall conform in the same way as for a *variable* and *expr* in an intrinsic assignment statement (7.4.1.2), as specified in Table 7.8. If necessary, each value of intrinsic type is converted according to the rules of intrinsic assignment (7.4.1.3) to a value that agrees in type and type parameters with the corresponding component of the derived type. For a nonpointer nonallocatable component, the shape of the expression shall conform with the shape of the component.

If a component with default initialization has no corresponding *component-data-source*, then the default initialization is applied to that component.

NOTE 4.57

Because no parent components appear in the defined component ordering, a value for a parent component may be specified only with a component keyword. Examples of equivalent values using types defined in Note 4.54:

```
! Create values with components x = 1.0, y = 2.0, color = 3.
TYPE(POINT) :: PV = POINT(1.0, 2.0)        ! Assume components of TYPE(POINT)
                                           ! are accessible here.
...
COLOR_POINT( point=point(1,2), color=3)    ! Value for parent component
COLOR_POINT( point=PV, color=3)            ! Available even if TYPE(point)
                                           ! has private components
COLOR_POINT( 1, 2, 3)                      ! All components of TYPE(point)
                                           ! need to be accessible.
```

A structure constructor shall not appear before the referenced type is defined.

NOTE 4.58

This example illustrates a derived-type constant expression using a derived type defined in Note 4.17:

```
PERSON (21, 'JOHN SMITH')
```

This could also be written as

```
PERSON (NAME = 'JOHN SMITH', AGE = 21)
```

NOTE 4.59

An example constructor using the derived type GENERAL_POINT defined in Note 4.24 is

```
general_point(dim=3) ( (/ 1., 2., 3. /) )
```

For a pointer component, the corresponding *component-data-source* shall be an allowable *data-target* or *proc-target* for such a pointer in a pointer assignment statement (7.4.2). If the component data source is a pointer, the association of the component is that of the pointer; otherwise, the component is pointer associated with the component data source.

NOTE 4.60

For example, if the variable TEXT were declared (5.1) to be

```
CHARACTER, DIMENSION (1:400), TARGET :: TEXT
```

and BIBLIO were declared using the derived-type definition REFERENCE in Note 4.31

```
TYPE (REFERENCE) :: BIBLIO
```

the statement

```
BIBLIO = REFERENCE (1, 1987, 1, "This is the title of the referenced &
                                &paper", TEXT)
```

is valid and associates the pointer component SYNOPSIS of the object BIBLIO with the target object TEXT.

If a component of a derived type is allocatable, the corresponding constructor expression shall either be a reference to the intrinsic function NULL with no arguments, an allocatable entity of the same rank, or shall evaluate to an entity of the same rank. If the expression is a reference to the intrinsic function NULL, the corresponding component of the constructor has a status of unallocated. If the expression is an allocatable entity, the corresponding component of the constructor has the same allocation status as that allocatable entity and, if it is allocated, the same dynamic type, bounds, and value; if a length parameter of the component is deferred, its value is the same as the corresponding parameter of the expression. Otherwise the corresponding component of the constructor has an allocation status of allocated and has the same bounds and value as the expression.

NOTE 4.61

When the constructor is an actual argument, the allocation status of the allocatable component is available through the associated dummy argument.

4.5.10 Derived-type operations and assignment

Intrinsic assignment of derived-type entities is described in 7.4.1. This standard does not specify any intrinsic operations on derived-type entities. Any operation on derived-type entities or defined assignment (7.4.1.4) for derived-type entities shall be defined explicitly by a function or a subroutine, and a generic interface (4.5.1, 12.3.2.1).

4.6 Enumerations and enumerators

An enumeration is a set of enumerators. An enumerator is a named integer constant. An enumeration definition specifies the enumeration and its set of enumerators of the corresponding integer kind.

R460	*enum-def*	**is**	*enum-def-stmt*
			enumerator-def-stmt
			[*enumerator-def-stmt*] ...
			end-enum-stmt
R461	*enum-def-stmt*	**is**	ENUM, BIND(C)
R462	*enumerator-def-stmt*	**is**	ENUMERATOR [::] *enumerator-list*
R463	*enumerator*	**is**	*named-constant* [= *scalar-int-initialization-expr*]
R464	*end-enum-stmt*	**is**	END ENUM

C492 (R462) If = appears in an *enumerator*, a double-colon separator shall appear before the *enumerator-list*.

For an enumeration, the kind is selected such that an integer type with that kind is interoperable (15.2.1) with the corresponding C enumeration type. The corresponding C enumeration type is the type that would be declared by a C enumeration specifier (6.7.2.2 of the C standard) that specified C enumeration constants with the same values as those specified by the *enum-def*, in the same order as specified by the *enum-def*.

The companion processor (2.5.10) shall be one that uses the same representation for the types declared by all C enumeration specifiers that specify the same values in the same order.

NOTE 4.62

If a companion processor uses an unsigned type to represent a given enumeration type, the Fortran processor will use the signed integer type of the same width for the enumeration, even though some of the values of the enumerators cannot be represented in this signed integer type. The values of any such enumerators will be interoperable with the values declared in the C enumeration.

NOTE 4.63

The C standard guarantees the enumeration constants fit in a C int (6.7.2.2 of the C standard). Therefore, the Fortran processor can evaluate all enumerator values using the integer type with kind parameter C_INT, and then determine the kind parameter of the integer type that is interoperable with the corresponding C enumerated type.

NOTE 4.64

The C standard specifies that two enumeration types are compatible only if they specify enumeration constants with the same names and same values in the same order. This standard further requires that a C processor that is to be a companion processor of a Fortran processor use the same representation for two enumeration types if they both specify enumeration constants with the same values in the same order, even if the names are different.

An enumerator is treated as if it were explicitly declared with the PARAMETER attribute. The enumerator is defined in accordance with the rules of intrinsic assignment (7.4) with the value determined as follows:

(1) If *scalar-int-initialization-expr* is specified, the value of the enumerator is the result of *scalar-int-initialization-expr*.

(2) If *scalar-int-initialization-expr* is not specified and the enumerator is the first enumerator in *enum-def*, the enumerator has the value 0.

(3) If *scalar-int-initialization-expr* is not specified and the enumerator is not the first enumerator in *enum-def*, its value is the result of adding 1 to the value of the enumerator that immediately precedes it in the *enum-def*.

NOTE 4.65

Example of an enumeration definition:

```
ENUM, BIND(C)
  ENUMERATOR :: RED = 4, BLUE = 9
  ENUMERATOR YELLOW
END ENUM
```

The kind type parameter for this enumeration is processor dependent, but the processor is required to select a kind sufficient to represent the values 4, 9, and 10, which are the values of its enumerators. The following declaration might be equivalent to the above enumeration definition.

```
INTEGER(SELECTED_INT_KIND(2)), PARAMETER :: RED = 4, BLUE = 9, YELLOW = 10
```

An entity of the same kind type parameter value can be declared using the intrinsic function KIND with one of the enumerators as its argument, for example

```
INTEGER(KIND(RED)) :: X
```

NOTE 4.66

There is no difference in the effect of declaring the enumerators in multiple ENUMERATOR statements or in a single ENUMERATOR statement. The order in which the enumerators in an enumeration definition are declared is significant, but the number of ENUMERATOR statements is not.

4.7 Construction of array values

An **array constructor** is defined as a sequence of scalar values and is interpreted as a rank-one array where the element values are those specified in the sequence.

R465	*array-constructor*	**is**	(/ *ac-spec* /)
		or	*left-square-bracket ac-spec right-square-bracket*
R466	*ac-spec*	**is**	*type-spec* ::
		or	[*type-spec* ::] *ac-value-list*
R467	*left-square-bracket*	**is**	[
R468	*right-square-bracket*	**is**	]
R469	*ac-value*	**is**	*expr*
		or	*ac-implied-do*
R470	*ac-implied-do*	**is**	(*ac-value-list* , *ac-implied-do-control*)
R471	*ac-implied-do-control*	**is**	*ac-do-variable* = *scalar-int-expr* , *scalar-int-expr* ■ ■ [, *scalar-int-expr*]
R472	*ac-do-variable*	**is**	*scalar-int-variable*

C493 (R472) *ac-do-variable* shall be a named variable.

C494 (R466) If *type-spec* is omitted, each *ac-value* expression in the *array-constructor* shall have the same type and kind type parameters.

C495 (R466) If *type-spec* specifies an intrinsic type, each *ac-value* expression in the *array-constructor* shall be of an intrinsic type that is in type conformance with a variable of type *type-spec* as specified in Table 7.8.

C496 (R466) If *type-spec* specifies a derived type, all *ac-value* expressions in the *array-constructor* shall be of that derived type and shall have the same kind type parameter values as specified by *type-spec*.

C497 (R470) The *ac-do-variable* of an *ac-implied-do* that is in another *ac-implied-do* shall not appear as the *ac-do-variable* of the containing *ac-implied-do*.

If *type-spec* is omitted, the type and type parameters of the array constructor are those of the *ac-value* expressions.

If *type-spec* appears, it specifies the type and type parameters of the array constructor. Each *ac-value* expression in the *array-constructor* shall be compatible with intrinsic assignment to a variable of this type and type parameters. Each value is converted to the type parameters of the *array-constructor* in accordance with the rules of intrinsic assignment (7.4.1.3).

The character length of an *ac-value* in an *ac-implied-do* whose iteration count is zero shall not depend on the value of the *ac-do-variable* and shall not depend on the value of an expression that is not an initialization expression.

If an *ac-value* is a scalar expression, its value specifies an element of the array constructor. If an *ac-value* is an array expression, the values of the elements of the expression, in array element order (6.2.2.2), specify the corresponding sequence of elements of the array constructor. If an *ac-value* is an *ac-implied-do*, it is expanded to form a sequence of elements under the control of the *ac-do-variable*, as in the DO construct (8.1.6.4).

For an *ac-implied-do*, the loop initialization and execution is the same as for a DO construct.

An empty sequence forms a zero-sized rank-one array.

NOTE 4.67

A one-dimensional array may be reshaped into any allowable array shape using the RESHAPE intrinsic function (13.7.99). An example is:

```
X = (/ 3.2, 4.01, 6.5 /)
Y = RESHAPE (SOURCE = [ 2.0, [ 4.5, 4.5 ], X ], SHAPE = [ 3, 2 ])
```

This results in Y having the 3×2 array of values:

```
2.0   3.2
4.5   4.01
4.5   6.5
```

NOTE 4.68

Examples of array constructors containing an implied DO are:

```
(/ (I, I = 1, 1075) /)
```

and

```
[ 3.6, (3.6 / I, I = 1, N) ]
```

NOTE 4.69

Using the type definition for PERSON in Note 4.17, an example of the construction of a derived-type array value is:

```
(/ PERSON (40, 'SMITH'), PERSON (20, 'JONES') /)
```

NOTE 4.70

Using the type definition for LINE in Note 4.28, an example of the construction of a derived-type scalar value with a rank-2 array component is:

```
LINE (RESHAPE ( (/ 0.0, 0.0, 1.0, 2.0 /), (/ 2, 2 /) ), 0.1, 1)
```

The RESHAPE intrinsic function is used to construct a value that represents a solid line from (0, 0) to (1, 2) of width 0.1 centimeters.

NOTE 4.71

Examples of zero-size array constructors are:

```
   (/ INTEGER :: /)
   (/ ( I, I = 1, 0) /)
```

NOTE 4.72

An example of an array constructor that specifies a length type parameter:

```
   (/ CHARACTER(LEN=7) :: 'Takata', 'Tanaka', 'Hayashi' /)
```

In this constructor, without the type specification, it would have been necessary to specify all of the constants with the same character length.

Section 5: Data object declarations and specifications

Every data object has a type and rank and may have type parameters and other attributes that determine the uses of the object. Collectively, these properties are the **attributes** of the object. The type of a named data object is either specified explicitly in a type declaration statement or determined implicitly by the first letter of its name (5.3). All of its attributes may be included in a type declaration statement or may be specified individually in separate specification statements.

NOTE 5.1

For example:

```
INTEGER :: INCOME, EXPENDITURE
```

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

```
REAL, DIMENSION (-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 therefore a size of 11.

5.1 Type declaration statements

R501	*type-declaration-stmt*	**is**	*declaration-type-spec* [[, *attr-spec*] ... ::] *entity-decl-list*
R502	*declaration-type-spec*	**is**	*intrinsic-type-spec*
		or	TYPE (*derived-type-spec*)
		or	CLASS (*derived-type-spec*)
		or	CLASS (*)

C501 (R502) In a *declaration-type-spec*, every *type-param-value* that is not a colon or an asterisk shall be a *specification-expr*.

C502 (R502) In a *declaration-type-spec* that uses the CLASS keyword, *derived-type-spec* shall specify an extensible type.

NOTE 5.2

A *declaration-type-spec* is used in a nonexecutable statement; a *type-spec* is used in an array constructor, a SELECT TYPE construct, or an ALLOCATE statement.

C503 (R502) The TYPE(*derived-type-spec*) shall not specify an abstract type (4.5.6).

R503	*attr-spec*	**is**	*access-spec*
		or	ALLOCATABLE
		or	ASYNCHRONOUS
		or	DIMENSION (*array-spec*)
		or	EXTERNAL
		or	INTENT (*intent-spec*)
		or	INTRINSIC
		or	*language-binding-spec*

		or	OPTIONAL
		or	PARAMETER
		or	POINTER
		or	PROTECTED
		or	SAVE
		or	TARGET
		or	VALUE
		or	VOLATILE
R504	*entity-decl*	**is**	*object-name* [(*array-spec*)] [* *char-length*] [*initialization*]
		or	*function-name* [* *char-length*]

C504 (R504) If a *type-param-value* in a *char-length* in an *entity-decl* is not a colon or an asterisk, it shall be a *specification-expr*.

R505	*object-name*	**is**	*name*

C505 (R505) The *object-name* shall be the name of a data object.

R506	*initialization*	**is**	= *initialization-expr*
		or	=> *null-init*
R507	*null-init*	**is**	*function-reference*

C506 (R507) The *function-reference* shall be a reference to the NULL intrinsic function with no arguments.

C507 (R501) The same *attr-spec* shall not appear more than once in a given *type-declaration-stmt*.

C508 An entity shall not be explicitly given any attribute more than once in a scoping unit.

C509 (R501) An entity declared with the CLASS keyword shall be a dummy argument or have the ALLOCATABLE or POINTER attribute.

C510 (R501) An array that has the POINTER or ALLOCATABLE attribute shall be specified with an *array-spec* that is a *deferred-shape-spec-list* (5.1.2.5.3).

C511 (R501) An *array-spec* for an *object-name* that is a function result that does not have the ALLOCATABLE or POINTER attribute shall be an *explicit-shape-spec-list*.

C512 (R501) If the POINTER attribute is specified, the ALLOCATABLE, TARGET, EXTERNAL, or INTRINSIC attribute shall not be specified.

C513 (R501) If the TARGET attribute is specified, the POINTER, EXTERNAL, INTRINSIC, or PARAMETER attribute shall not be specified.

C514 (R501) The PARAMETER attribute shall not be specified for a dummy argument, a pointer, an allocatable entity, a function, or an object in a common block.

C515 (R501) The INTENT, VALUE, and OPTIONAL attributes may be specified only for dummy arguments.

C516 (R501) The INTENT attribute shall not be specified for a dummy procedure without the POINTER attribute.

C517 (R501) The SAVE attribute shall not be specified for an object that is in a common block, a dummy argument, a procedure, a function result, an automatic data object, or an object with the PARAMETER attribute.

C518 An entity shall not have both the EXTERNAL attribute and the INTRINSIC attribute.

C519 (R501) An entity in an *entity-decl-list* shall not have the EXTERNAL or INTRINSIC attribute specified unless it is a function.

C520 (R504) The * *char-length* option is permitted only if the type specified is character.

C521 (R504) The *function-name* shall be the name of an external function, an intrinsic function, a function dummy procedure, or a statement function.

C522 (R501) The *initialization* shall appear if the statement contains a PARAMETER attribute (5.1.2.10).

C523 (R501) If *initialization* appears, a double-colon separator shall appear before the *entity-decl-list*.

C524 (R504)*initialization* shall not appear if *object-name* is a dummy argument, a function result, an object in a named common block unless the type declaration is in a block data program unit, an object in blank common, an allocatable variable, an external name, an intrinsic name, or an automatic object.

C525 (R504) If => appears in *initialization*, the object shall have the POINTER attribute. If = appears in *initialization*, the object shall not have the POINTER attribute.

C526 (R501) If the VOLATILE attribute is specified, the PARAMETER, INTRINSIC, EXTERNAL, or INTENT(IN) attribute shall not be specified.

C527 (R501) If the VALUE attribute is specified, the PARAMETER, EXTERNAL, POINTER, ALLOCATABLE, DIMENSION, VOLATILE, INTENT(INOUT), or INTENT(OUT) attribute shall not be specified.

C528 (R501) If the VALUE attribute is specified for a dummy argument of type character, the length parameter shall be omitted or shall be specified by an initialization expression with the value one.

C529 (R501) The VALUE attribute shall not be specified for a dummy procedure.

C530 (R501) The ALLOCATABLE, POINTER, or OPTIONAL attribute shall not be specified for a dummy argument of a procedure that has a *proc-language-binding-spec*.

C531 (R503) A *language-binding-spec* shall appear only in the specification part of a module.

C532 (R501) If a *language-binding-spec* is specified, the entity declared shall be an interoperable variable (15.2).

C533 (R501) If a *language-binding-spec* with a NAME= specifier appears, the *entity-decl-list* shall consist of a single *entity-decl*.

C534 (R503) The PROTECTED attribute is permitted only in the specification part of a module.

C535 (R501) The PROTECTED attribute is permitted only for a procedure pointer or named variable that is not in a common block.

C536 (R501) If the PROTECTED attribute is specified, the EXTERNAL, INTRINSIC, or PARAMETER attribute shall not be specified.

C537 A nonpointer object that has the PROTECTED attribute and is accessed by use association shall not appear in a variable definition context (16.5.7) or as the *data-target* or *proc-target* in a *pointer-assignment-stmt*.

C538 A pointer object that has the PROTECTED attribute and is accessed by use association shall not appear as

(1) A *pointer-object* in a *pointer-assignment-stmt* or *nullify-stmt*,

(2) An *allocate-object* in an *allocate-stmt* or *deallocate-stmt*, or

(3) An actual argument in a reference to a procedure if the associated dummy argument is a pointer with the INTENT(OUT) or INTENT(INOUT) attribute.

A name that identifies a specific intrinsic function in a scoping unit has a type as specified in 13.6. An explicit type declaration statement is not required; however, it is permitted. Specifying a type for a generic intrinsic function name in a type declaration statement is not sufficient, by itself, to remove the generic properties from that function.

A function result may be declared to have the POINTER or ALLOCATABLE attribute.

A *specification-expr* in an *array-spec*, in a *type-param-value* in a *declaration-type-spec* corresponding to a length type parameter, or in a *char-length* in an *entity-decl* shall be an initialization expression unless it is in an interface body (12.3.2.1), the specification part of a subprogram, or the *declaration-type-spec* of a FUNCTION statement (12.5.2.1). If the data object being declared depends on the value of a *specification-expr* that is not an initialization expression, and it is not a dummy argument, such an object is called an **automatic data object**.

NOTE 5.3

An automatic object shall neither appear in a SAVE or DATA statement nor be declared with a SAVE attribute nor be initially defined by an *initialization*.

If a type parameter in a *declaration-type-spec* or in a *char-length* in an *entity-decl* is defined by an expression that is not an initialization expression, the type parameter value is established on entry to the procedure and is not affected by any redefinition or undefinition of the variables in the specification expression during execution of the procedure.

If an *entity-decl* contains *initialization* and the *object-name* does not have the PARAMETER attribute, the entity is a variable with **explicit initialization**. Explicit initialization alternatively may be specified in a DATA statement unless the variable is of a derived type for which default initialization is specified. If *initialization* is =*initialization-expr*, the *object-name* is initially defined with the value specified by the *initialization-expr*; if necessary, the value is converted according to the rules of intrinsic assignment (7.4.1.3) to a value that agrees in type, type parameters, and shape with the *object-name*. A variable, or part of a variable, shall not be explicitly initialized more than once in a program. If the variable is an array, it shall have its shape specified in either the type declaration statement or a previous attribute specification statement in the same scoping unit.

If *initialization* is =>*null-init*, *object-name* shall be a pointer, and its initial association status is disassociated.

The presence of *initialization* implies that *object-name* is saved, except for an *object-name* in a named common block or an *object-name* with the PARAMETER attribute. The implied SAVE attribute may be reaffirmed by explicit use of the SAVE attribute in the type declaration statement, by inclusion of the *object-name* in a SAVE statement (5.2.12), or by the appearance of a SAVE statement without a *saved-entity-list* in the same scoping unit.

NOTE 5.4

Examples of type declaration statements are:

NOTE 5.4 (cont.)

```
REAL A (10)
LOGICAL, DIMENSION (5, 5) :: MASK1, MASK2
COMPLEX :: CUBE_ROOT = (-0.5, 0.866)
INTEGER, PARAMETER :: SHORT = SELECTED_INT_KIND (4)
INTEGER (SHORT) K     !  Range at least -9999 to 9999.
REAL (KIND (0.0D0)) A
REAL (KIND = 2) B
COMPLEX (KIND = KIND (0.0D0)) :: C
CHARACTER (LEN = 10, KIND = 2)  A
CHARACTER B, C *20
TYPE (PERSON) :: CHAIRMAN
TYPE(NODE), POINTER :: HEAD => NULL ( )
TYPE (humongous_matrix (k=8, d=1000)) :: mat
```

(The last line above uses a type definition from Note 4.24.)

5.1.1 Declaration type specifiers

The *declaration-type-spec* in a type declaration statement specifies the type of the entities in the entity declaration list. This explicit type declaration may override or confirm the implicit type that could otherwise be indicated by the first letter of an entity name (5.3).

An *intrinsic-type-spec* in a type declaration statement is used to declare entities of intrinsic type.

5.1.1.1 TYPE

A TYPE type specifier is used to declare entities of a derived type.

Where a data entity is declared explicitly using the TYPE type specifier, the specified derived type shall have been defined previously in the scoping unit or be accessible there by use or host association. If the data entity is a function result, the derived type may be specified in the FUNCTION statement provided the derived type is defined within the body of the function or is accessible there by use or host association. If the derived type is specified in the FUNCTION statement and is defined within the body of the function, it is as if the function result variable was declared with that derived type immediately following the *derived-type-def* of the specified derived type.

A scalar entity of derived type is a **structure**. If a derived type has the SEQUENCE property, a scalar entity of the type is a **sequence structure**.

5.1.1.2 CLASS

A **polymorphic** entity is a data entity that is able to be of differing types during program execution. The type of a data entity at a particular point during execution of a program is its **dynamic type**. The **declared type** of a data entity is the type that it is declared to have, either explicitly or implicitly.

A CLASS type specifier is used to declare polymorphic objects. The declared type of a polymorphic object is the specified type if the CLASS type specifier contains a type name.

An object declared with the CLASS(*) specifier is an **unlimited polymorphic** object. An unlimited polymorphic entity is not declared to have a type. It is not considered to have the same declared type as any other entity, including another unlimited polymorphic entity.

A nonpolymorphic entity is **type compatible** only with entities of the same type. For a polymorphic entity, type compatibility is based on its declared type. A polymorphic entity that is not an unlimited polymorphic entity is type compatible with entities of the same type or any of its extensions. Even though an unlimited polymorphic entity is not considered to have a declared type, it is type compatible with all entities. An entity is said to be type compatible with a type if it is type compatible with entities of that type.

Two entities are **type incompatible** if neither is type compatible with the other.

An entity is type, kind, and rank compatible, or **TKR compatible**, with another entity if the first entity is type compatible with the second, the kind type parameters of the first entity have the same values as corresponding kind type parameters of the second, and both entities have the same rank.

A polymorphic allocatable object may be allocated to be of any type with which it is type compatible. A polymorphic pointer or dummy argument may, during program execution, be associated with objects with which it is type compatible.

The dynamic type of an allocated allocatable polymorphic object is the type with which it was allocated. The dynamic type of an associated polymorphic pointer is the dynamic type of its target. The dynamic type of a nonallocatable nonpointer polymorphic dummy argument is the dynamic type of its associated actual argument. The dynamic type of an unallocated allocatable or a disassociated pointer is the same as its declared type. The dynamic type of an entity identified by an associate name (8.1.4) is the dynamic type of the selector with which it is associated. The dynamic type of an object that is not polymorphic is its declared type.

NOTE 5.5

Only components of the declared type of a polymorphic object may be designated by component selection (6.1.2).

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 entities being declared or specify restrictions on their use in the program.

5.1.2.1 Accessibility attribute

The **accessibility attribute** specifies the accessibility of an entity via a particular identifier.

R508	*access-spec*	**is**	PUBLIC
		or	PRIVATE

C539 (R508) An *access-spec* shall appear only in the *specification-part* of a module.

Identifiers that are specified in a module or accessible in that module by use association have either the PUBLIC or PRIVATE attribute. Identifiers for which an *access-spec* is not explicitly specified in that module have the default accessibility attribute for that module. The default accessibility attribute for a module is PUBLIC unless it has been changed by a PRIVATE statement (5.2.1). Only identifiers that have the PUBLIC attribute in that module are available to be accessed from that module by use association.

NOTE 5.6

In order for an identifier to be accessed by use association, it must have the PUBLIC attribute in the module from which it is accessed. It can nonetheless have the PRIVATE attribute in a module in which it is accessed by use association, and therefore not be available for use association from a module where it is PRIVATE.

NOTE 5.7

An example of an accessibility specification is:

```
REAL, PRIVATE :: X, Y, Z
```

5.1.2.2 ALLOCATABLE attribute

An object with the **ALLOCATABLE attribute** is one for which space is allocated by an ALLOCATE statement (6.3.1) or by a derived-type intrinsic assignment statement (7.4.1.3).

5.1.2.3 ASYNCHRONOUS attribute

The **ASYNCHRONOUS attribute** specifies that a variable may be subject to asynchronous input/output.

The base object of a variable shall have the ASYNCHRONOUS attribute in a scoping unit if:

(1) the variable appears in an executable statement or specification expression in that scoping unit and

(2) any statement of the scoping unit is executed while the variable is a pending I/O storage sequence affector (9.5.1.4)

The ASYNCHRONOUS attribute may be conferred implicitly by the use of a variable in an asynchronous input/output statement (9.5.1.4).

An object may have the ASYNCHRONOUS attribute in a particular scoping unit without necessarily having it in other scoping units. If an object has the ASYNCHRONOUS attribute, then all of its subobjects also have the ASYNCHRONOUS attribute.

NOTE 5.8

The ASYNCHRONOUS attribute specifies the variables that might be associated with a pending input/output storage sequence (the actual memory locations on which asynchronous input/output is being performed) while the scoping unit is in execution. This information could be used by the compiler to disable certain code motion optimizations.

The ASYNCHRONOUS attribute is similar to the VOLATILE attribute. It is intended to facilitate traditional code motion optimizations in the presence of asynchronous input/output.

5.1.2.4 BIND attribute for data entities

The BIND attribute for a variable or common block specifies that it is capable of interoperating with a C variable that has external linkage (15.3).

R509 *language-binding-spec* **is** BIND (C [, NAME = *scalar-char-initialization-expr*])

C540 (R509) The *scalar-char-initialization-expr* shall be of default character kind.

NOTE 5.9

The C standard provides a facility for creating C identifiers whose characters are not restricted to the C basic character set. Such a C identifier is referred to as a universal character name (6.4.3 of the C standard). The name of such a C identifier may include characters that are not part of the representation method used by the processor for type default character. If so, the C entity cannot be linked (12.5.3, 15.3.1) with a Fortran entity.

This standard does not require a processor to provide a means of linking Fortran entities with C

NOTE 5.9 (cont.)

entities whose names are specified using the universal character name facility.

The BIND attribute implies the SAVE attribute, which may be confirmed by explicit specification.

NOTE 5.10

Specifying the BIND attribute for an entity might have no discernable effect for a processor that is its own companion processor.

5.1.2.5 DIMENSION attribute

The **DIMENSION attribute** specifies that an entity is an array. The rank or rank and shape is specified by the *array-spec*, if there is one, in the *entity-decl*, or by the *array-spec* in the DIMENSION *attr-spec* otherwise. To specify that an entity is an array in a type declaration statement, either the DIMENSION *attr-spec* shall appear, or an *array-spec* shall appear in the *entity-decl*. The appearance of an *array-spec* in an *entity-decl* specifies the DIMENSION attribute for the entity. The DIMENSION attribute alternatively may be specified in the specification statements DIMENSION, ALLOCATABLE, POINTER, TARGET, or COMMON.

R510	*array-spec*	**is**	*explicit-shape-spec-list*
		or	*assumed-shape-spec-list*
		or	*deferred-shape-spec-list*
		or	*assumed-size-spec*

C541 (R510)The maximum rank is seven.

NOTE 5.11

Examples of DIMENSION attribute specifications are:

```
SUBROUTINE EX (N, A, B)
   REAL, DIMENSION (N, 10) :: W          ! Automatic explicit-shape array
   REAL A (:), B (0:)                    ! Assumed-shape arrays
   REAL, POINTER :: D (:, :)             ! Array pointer
   REAL, DIMENSION (:), POINTER :: P     ! Array pointer
   REAL, ALLOCATABLE, DIMENSION (:) :: E ! Allocatable array
```

5.1.2.5.1 Explicit-shape array

An **explicit-shape array** is a named array that is declared with an *explicit-shape-spec-list*. This specifies explicit values for the bounds in each dimension of the array.

R511	*explicit-shape-spec*	**is**	[*lower-bound* :] *upper-bound*
R512	*lower-bound*	**is**	*specification-expr*
R513	*upper-bound*	**is**	*specification-expr*

C542 (R511) An explicit-shape array whose bounds are not initialization expressions shall be a dummy argument, a function result, or an automatic array of a procedure.

An **automatic array** is an explicit-shape array that is declared in a subprogram, is not a dummy argument, and has bounds that are not initialization expressions.

If an explicit-shape array has bounds that are not initialization expressions, the bounds, and hence shape, are determined at entry to the procedure by evaluating the bounds expressions. The bounds of such an array are unaffected by the redefinition or undefinition of any variable during execution of the procedure.

The values of each *lower-bound* and *upper-bound* determine the bounds of the array along a particular dimension and hence the extent of the array in that dimension. The value of a lower bound or an upper bound may be positive, negative, or zero. The 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 default value is 1. The number of sets of bounds specified is the rank.

5.1.2.5.2 Assumed-shape array

An **assumed-shape array** is a nonpointer dummy argument array that takes its shape from the associated actual argument array.

R514 *assumed-shape-spec* **is** [*lower-bound*] :

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

The extent of a dimension of an assumed-shape array dummy argument is the extent of the corresponding dimension of the associated actual argument array. If the lower bound value is d and the extent of the corresponding dimension of the associated actual argument array is s, then the value of the upper bound is $s + d - 1$. The lower bound is *lower-bound*, if present, and 1 otherwise.

5.1.2.5.3 Deferred-shape array

A **deferred-shape array** is an allocatable array or an array pointer.

An **allocatable array** is an array that has the ALLOCATABLE attribute and a specified rank, but its bounds, and hence shape, are determined by allocation or argument association.

An array with the ALLOCATABLE attribute shall be declared with a *deferred-shape-spec-list*.

An **array pointer** is an array with the POINTER attribute and a specified rank. Its bounds, and hence shape, are determined when it is associated with a target. An array with the POINTER attribute shall be declared with a *deferred-shape-spec-list*.

R515 *deferred-shape-spec* **is** :

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

The size, bounds, and shape of an unallocated allocatable array or a disassociated array pointer are undefined. No part of such an array shall be referenced or defined; however, the array may appear as an argument to an intrinsic inquiry function as specified in 13.1.

The bounds of each dimension of an allocatable array are those specified when the array is allocated.

The bounds of each dimension of an array pointer may be specified in two ways:

(1) in an ALLOCATE statement (6.3.1) when the target is allocated, or
(2) by pointer assignment (7.4.2).

The bounds of the array target or allocatable array are unaffected by any subsequent redefinition or undefinition of variables involved in the bounds' specification expressions.

5.1.2.5.4 Assumed-size array

An **assumed-size array** is a dummy argument array whose 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. An assumed-size array is declared with an *assumed-size-spec*.

R516 *assumed-size-spec* **is** [*explicit-shape-spec-list* ,] [*lower-bound* :] *

C543 An *assumed-size-spec* shall not appear except as the declaration of the array bounds of a dummy data argument.

C544 An assumed-size array with INTENT (OUT) shall not be of a type for which default initialization is specified.

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 of any type other than default 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 default character with a subscript order value of r (6.2.2.2) in an array of size x, the size of the dummy array is $x - r + 1$.

(3) If the actual argument is a default character array, default character array element, or a default character array element substring (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.

(4) If the actual argument is of type default character and is a scalar that is not an array element or array element substring designator, the size of the dummy array is MAX (INT (l/e), 0), where e is the length of an element in the dummy character array and l is the length of the actual argument.

The rank equals one plus the number of *explicit-shape-spec*s.

An assumed-size array has no upper bound in its last dimension and therefore has no extent in its last dimension and no shape. An assumed-size array name shall not be written as a whole array reference except as an actual argument in a procedure reference for which the shape is not required.

The bounds of the first $n - 1$ dimensions are those specified by the *explicit-shape-spec-list*, if present, in the *assumed-size-spec*. The lower bound of the last dimension is *lower-bound*, if present, and 1 otherwise. An assumed-size array may be subscripted or sectioned (6.2.2.3). The upper bound shall not be omitted from a subscript triplet in the last dimension.

If an assumed-size array has bounds that are not initialization expressions, the bounds are determined at entry to the procedure. The bounds of such an array are unaffected by the redefinition or undefinition of any variable during execution of the procedure.

5.1.2.6 EXTERNAL attribute

The **EXTERNAL attribute** specifies that an entity is an external procedure, dummy procedure, procedure pointer, or block data subprogram. This attribute may also be specified by an EXTERNAL statement (12.3.2.2), a *procedure-declaration-stmt* (12.3.2.3) or an interface body that is not in an abstract interface block (12.3.2.1).

If an external procedure or dummy procedure is used as an actual argument or is the target of a procedure pointer assignment, it shall be declared to have the EXTERNAL attribute.

A procedure that has both the EXTERNAL and POINTER attributes is a procedure pointer.

5.1.2.7 INTENT attribute

The **INTENT attribute** specifies the intended use of a dummy argument.

R517	*intent-spec*	**is**	IN
		or	OUT
		or	INOUT

C545 (R517) A nonpointer object with the INTENT (IN) attribute shall not appear in a variable definition context (16.5.7).

C546 (R517) A pointer object with the INTENT (IN) attribute shall not appear as

(1) A *pointer-object* in a *pointer-assignment-stmt* or *nullify-stmt*,
(2) An *allocate-object* in an *allocate-stmt* or *deallocate-stmt*, or
(3) An actual argument in a reference to a procedure if the associated dummy argument is a pointer with the INTENT (OUT) or INTENT (INOUT) attribute.

The INTENT (IN) attribute for a nonpointer dummy argument specifies that it shall neither be defined nor become undefined during the execution of the procedure. The INTENT (IN) attribute for a pointer dummy argument specifies that during the execution of the procedure its association shall not be changed except that it may become undefined if the target is deallocated other than through the pointer (16.4.2.1.3).

The INTENT (OUT) attribute for a nonpointer dummy argument specifies that it shall be defined before a reference to the dummy argument is made within the procedure and any actual argument that becomes associated with such a dummy argument shall be definable. On invocation of the procedure, such a dummy argument becomes undefined except for components of an object of derived type for which default initialization has been specified. The INTENT (OUT) attribute for a pointer dummy argument specifies that on invocation of the procedure the pointer association status of the dummy argument becomes undefined. Any actual argument associated with such a pointer dummy shall be a pointer variable.

The INTENT (INOUT) attribute for a nonpointer dummy argument specifies that it is intended for use both to receive data from and to return data to the invoking scoping unit. Such a dummy argument may be referenced or defined. Any actual argument that becomes associated with such a dummy argument shall be definable. The INTENT (INOUT) attribute for a pointer dummy argument specifies that it is intended for use both to receive a pointer association from and to return a pointer association to the invoking scoping unit. Any actual argument associated with such a pointer dummy shall be a pointer variable.

If no INTENT attribute is specified for a dummy argument, its use is subject to the limitations of the associated actual argument (12.4.1.2, 12.4.1.3, 12.4.1.4).

NOTE 5.12

An example of INTENT specification is:

```
SUBROUTINE MOVE (FROM, TO)
   USE PERSON_MODULE
   TYPE (PERSON), INTENT (IN) :: FROM
   TYPE (PERSON), INTENT (OUT) :: TO
```

If an object has an INTENT attribute, then all of its subobjects have the same INTENT attribute.

NOTE 5.13

If a dummy argument is a derived-type object with a pointer component, then the pointer as a pointer is a subobject of the dummy argument, but the target of the pointer is not. Therefore, the restrictions on subobjects of the dummy object apply to the pointer in contexts where it is used as a pointer, but not in contexts where it is dereferenced to indicate its target. For example, if X is a dummy argument of derived type with an integer pointer component P, and X has INTENT(IN), then the statement

```
X%P => NEW_TARGET
```

is prohibited, but

```
X%P = 0
```

is allowed (provided that X%P is associated with a definable target).

Similarly, the INTENT restrictions on pointer dummy arguments apply only to the association of the dummy argument; they do not restrict the operations allowed on its target.

NOTE 5.14

Argument intent specifications serve several purposes in addition to documenting the intended use of dummy arguments. A processor can check whether an INTENT (IN) dummy argument is used in a way that could redefine it. A slightly more sophisticated processor could check to see whether an INTENT (OUT) dummy argument could possibly be referenced before it is defined. If the procedure's interface is explicit, the processor can also verify that actual arguments corresponding to INTENT (OUT) or INTENT (INOUT) dummy arguments are definable. A more sophisticated processor could use this information to optimize the translation of the referencing scoping unit by taking advantage of the fact that actual arguments corresponding to INTENT (IN) dummy arguments will not be changed and that any prior value of an actual argument corresponding to an INTENT (OUT) dummy argument will not be referenced and could thus be discarded.

INTENT (OUT) means that the value of the argument after invoking the procedure is entirely the result of executing that procedure. If there is any possibility that an argument should retain its current value rather than being redefined, INTENT (INOUT) should be used rather than INTENT (OUT), even if there is no explicit reference to the value of the dummy argument. Because an INTENT(OUT) variable is considered undefined on entry to the procedure, any default initialization specified for its type will be applied.

INTENT (INOUT) is not equivalent to omitting the INTENT attribute. The argument corresponding to an INTENT (INOUT) dummy argument always shall be definable, while an argument corresponding to a dummy argument without an INTENT attribute need be definable only if the dummy argument is actually redefined.

5.1.2.8 INTRINSIC attribute

The **INTRINSIC attribute** confirms that a name is the specific name (13.6) or generic name (13.5) of an intrinsic procedure. The INTRINSIC attribute allows the specific name of an intrinsic procedure that is listed in 13.6 and not marked with a bullet (•) to be used as an actual argument (12.4).

Declaring explicitly that a generic intrinsic procedure name has the INTRINSIC attribute does not cause that name to lose its generic property.

If the specific name of an intrinsic procedure (13.6) is used as an actual argument, the name shall be explicitly specified to have the INTRINSIC attribute.

C547 (R503) (R1216) If the name of a generic intrinsic procedure is explicitly declared to have the INTRINSIC attribute, and it is also the generic name in one or more generic interfaces (12.3.2.1) accessible in the same scoping unit, the procedures in the interfaces and the specific intrinsic procedures shall all be functions or all be subroutines, and the characteristics of the specific intrinsic procedures and the procedures in the interfaces shall differ as specified in 16.2.3.

5.1.2.9 OPTIONAL attribute

The **OPTIONAL attribute** specifies that the dummy argument need not be associated with an actual argument in a reference to the procedure (12.4.1.6). The PRESENT intrinsic function may be used to determine whether an actual argument has been associated with a dummy argument having the OPTIONAL attribute.

5.1.2.10 PARAMETER attribute

The **PARAMETER attribute** specifies entities that are named constants. The *object-name* has the value specified by the *initialization-expr* that appears on the right of the equals; if necessary, the value is converted according to the rules of intrinsic assignment (7.4.1.3) to a value that agrees in type, type parameters, and shape with the *object-name*.

A named constant shall not be referenced unless it has been defined previously in the same statement, defined in a prior statement, or made accessible by use or host association.

NOTE 5.15

Examples of declarations with a PARAMETER attribute are:

```
REAL, PARAMETER :: ONE = 1.0, Y = 4.1 / 3.0
INTEGER, DIMENSION (3), PARAMETER :: ORDER = (/ 1, 2, 3 /)
TYPE(NODE), PARAMETER :: DEFAULT = NODE(0, NULL ( ))
```

5.1.2.11 POINTER attribute

Entities with the **POINTER attribute** can be associated with different data objects or procedures during execution of a program. A pointer is either a data pointer or a procedure pointer. Procedure pointers are described in 12.3.2.3.

A data pointer shall neither be referenced nor defined unless it is pointer associated with a target object that may be referenced or defined.

If a data pointer is associated, the values of its deferred type parameters are the same as the values of the corresponding type parameters of its target.

A procedure pointer shall not be referenced unless it is pointer associated with a target procedure.

NOTE 5.16

Examples of POINTER attribute specifications are:

```
TYPE (NODE), POINTER :: CURRENT, TAIL
REAL, DIMENSION (:, :), POINTER :: IN, OUT, SWAP
```

For a more elaborate example see C.2.1.

5.1.2.12 PROTECTED attribute

The **PROTECTED attribute** imposes limitations on the usage of module entities.

Other than within the module in which an entity is given the PROTECTED attribute,

(1) if it is a nonpointer object, it is not definable, and
(2) if it is a pointer, its association status shall not be changed except that it may become undefined if its target is deallocated other than through the pointer (16.4.2.1.3) or if its target becomes undefined by execution of a RETURN or END statement.

If an object has the PROTECTED attribute, all of its subobjects have the PROTECTED attribute.

NOTE 5.17

An example of the PROTECTED attribute:

```
MODULE temperature
  REAL, PROTECTED :: temp_c, temp_f
CONTAINS
  SUBROUTINE set_temperature_c(c)
    REAL, INTENT(IN) :: c
    temp_c = c
    temp_f = temp_c*(9.0/5.0) + 32
  END SUBROUTINE
END MODULE
```

The PROTECTED attribute ensures that the variables `temp_c` and `temp_f` cannot be modified other than via the `set_temperature_c` procedure, thus keeping them consistent with each other.

5.1.2.13 SAVE attribute

An entity with the **SAVE attribute**, in the scoping unit of a subprogram, retains its association status, allocation status, definition status, and value after execution of a RETURN or END statement unless it is a pointer and its target becomes undefined (16.4.2.1.3(4)). It is shared by all instances (12.5.2.3) of the subprogram.

An entity with the SAVE attribute, declared in the scoping unit of a module, retains its association status, allocation status, definition status, and value after a RETURN or END statement is executed in a procedure that accesses the module unless it is a pointer and its target becomes undefined.

A **saved** entity is an entity that has the SAVE attribute. An **unsaved** entity is an entity that does not have the SAVE attribute.

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

5.1.2.14 TARGET attribute

An object with the **TARGET attribute** may have a pointer associated with it (7.4.2). An object without the TARGET attribute shall not have an accessible pointer associated with it.

NOTE 5.18

In addition to variables explicitly declared to have the TARGET attribute, the objects created by allocation of pointers (6.3.1.2) have the TARGET attribute.

If an object has the TARGET attribute, then all of its nonpointer subobjects also have the TARGET attribute.

NOTE 5.19

Examples of TARGET attribute specifications are:

```
TYPE (NODE), TARGET :: HEAD
REAL, DIMENSION (1000, 1000), TARGET :: A, B
```

For a more elaborate example see C.2.2.

NOTE 5.20

Every object designator that starts from a target object will have either the TARGET or POINTER attribute. If pointers are involved, the designator might not necessarily be a subobject of the original target object, but because pointers may point only to targets, there is no way to end up at a nonpointer that is not a target.

5.1.2.15 VALUE attribute

The **VALUE attribute** specifies a type of argument association (12.4.1.2) for a dummy argument.

5.1.2.16 VOLATILE attribute

The **VOLATILE attribute** specifies that an object may be referenced, defined, or become undefined, by means not specified by the program.

An object may have the VOLATILE attribute in a particular scoping unit without necessarily having it in other scoping units. If an object has the VOLATILE attribute, then all of its subobjects also have the VOLATILE attribute.

NOTE 5.21

The Fortran processor should use the most recent definition of a volatile object when a value is required. Likewise, it should make the most recent Fortran definition available. It is the programmer's responsibility to manage the interactions with the non-Fortran processes.

A pointer with the VOLATILE attribute may additionally have its association status and array bounds changed by means not specified by the program.

NOTE 5.22

If the target of a pointer is referenced, defined, or becomes undefined, by means not specified by the program, while the pointer is associated with the target, then the pointer shall have the VOLATILE attribute. Usually a pointer should have the VOLATILE attribute if its target has the VOLATILE attribute. Similarly, all members of an EQUIVALENCE group should have the VOLATILE attribute if one member has the VOLATILE attribute.

An allocatable object with the VOLATILE attribute may additionally have its allocation status, dynamic type and type parameters, and array bounds changed by means not specified by the program.

5.2 Attribute specification statements

All attributes (other than type) may be specified for entities, independently of type, by separate attribute specification statements. The combination of attributes that may be specified for a particular entity is subject to the same restrictions as for type declaration statements regardless of the method of specification. This also applies to PROCEDURE, EXTERNAL, and INTRINSIC statements.

5.2.1 Accessibility statements

R518	*access-stmt*	**is**	*access-spec* [[::] *access-id-list*]
R519	*access-id*	**is**	*use-name*
		or	*generic-spec*

C548 (R518) An *access-stmt* shall appear only in the *specification-part* of a module. Only one accessibility statement with an omitted *access-id-list* is permitted in the *specification-part* of a module.

C549 (R519) Each *use-name* shall be the name of a named variable, procedure, derived type, named constant, or namelist group.

An *access-stmt* with an *access-id-list* specifies the accessibility attribute (5.1.2.1), PUBLIC or PRIVATE, of each *access-id* in the list. An *access-stmt* without an *access-id* list specifies the default accessibility that applies to all potentially accessible identifiers in the *specification-part* of the module. The statement

```
PUBLIC
```

specifies a default of public accessibility. The statement

```
PRIVATE
```

specifies a default of private accessibility. If no such statement appears in a module, the default is public accessibility.

NOTE 5.23

Examples of accessibility statements are:

```
MODULE EX
   PRIVATE
   PUBLIC :: A, B, C, ASSIGNMENT (=), OPERATOR (+)
```

5.2.2 ALLOCATABLE statement

R520	*allocatable-stmt*	**is**	ALLOCATABLE [::] ■
			■ *object-name* [(*deferred-shape-spec-list*)] ■
			■ [, *object-name* [(*deferred-shape-spec-list*)]] ...

This statement specifies the ALLOCATABLE attribute (5.1.2.2) for a list of objects.

NOTE 5.24

An example of an ALLOCATABLE statement is:

```
REAL A, B (:), SCALAR
ALLOCATABLE :: A (:, :), B, SCALAR
```

5.2.3 ASYNCHRONOUS statement

R521	*asynchronous-stmt*	**is**	ASYNCHRONOUS [::] *object-name-list*

The ASYNCHRONOUS statement specifies the ASYNCHRONOUS attribute (5.1.2.3) for a list of objects.

5.2.4 BIND statement

R522 *bind-stmt* **is** *language-binding-spec* [::] *bind-entity-list*
R523 *bind-entity* **is** *entity-name*
or / *common-block-name* /

C550 (R522) If any *bind-entity* in a *bind-stmt* is an *entity-name*, the *bind-stmt* shall appear in the specification part of a module and the entity shall be an interoperable variable (15.2.4, 15.2.5).

C551 (R522) If the *language-binding-spec* has a NAME= specifier, the *bind-entity-list* shall consist of a single *bind-entity*.

C552 (R522) If a *bind-entity* is a common block, each variable of the common block shall be interoperable (15.2.4, 15.2.5).

The BIND statement specifies the BIND attribute (5.1.2.4) for a list of variables and common blocks.

5.2.5 DATA statement

R524 *data-stmt* **is** DATA *data-stmt-set* [[,] *data-stmt-set*] ...

This statement is used to specify explicit initialization (5.1).

A variable, or part of a variable, shall not be explicitly initialized more than once in a program. If a nonpointer object has been specified with default initialization in a type definition, it shall not appear in a *data-stmt-object-list*.

A variable that appears in a DATA statement and has not been typed previously may appear in a subsequent type declaration only if that declaration confirms the implicit typing. An array name, array section, or array element that appears in a DATA statement shall have had its array properties established by a previous specification statement.

Except for variables in named common blocks, a named variable has the SAVE attribute if any part of it is initialized in a DATA statement, and this may be confirmed by a SAVE statement or a type declaration statement containing the SAVE attribute.

R525 *data-stmt-set* **is** *data-stmt-object-list* / *data-stmt-value-list* /
R526 *data-stmt-object* **is** *variable*
or *data-implied-do*
R527 *data-implied-do* **is** (*data-i-do-object-list* , *data-i-do-variable* = ■
■ *scalar-int-expr* , *scalar-int-expr* [, *scalar-int-expr*])
R528 *data-i-do-object* **is** *array-element*
or *scalar-structure-component*
or *data-implied-do*
R529 *data-i-do-variable* **is** *scalar-int-variable*

C553 (R526) In a *variable* that is a *data-stmt-object*, any subscript, section subscript, substring starting point, and substring ending point shall be an initialization expression.

C554 (R526) A variable whose designator is included in a *data-stmt-object-list* or a *data-i-do-object-list* shall not be: a dummy argument, made accessible by use association or host association, in a named common block unless the DATA statement is in a block data program unit, in a blank common block, a function name, a function result name, an automatic object, or an allocatable variable.

C555 (R526) A *data-i-do-object* or a *variable* that appears as a *data-stmt-object* shall not be an object designator in which a pointer appears other than as the entire rightmost *part-ref*.

C556 (R529) The *data-i-do-variable* shall be a named variable.

C557 (R527) A *scalar-int-expr* of a *data-implied-do* shall involve as primaries only constants, subobjects of constants, or DO variables of the containing *data-implied-do*s, and each operation shall be intrinsic.

C558 (R528) The *array-element* shall be a variable.

C559 (R528) The *scalar-structure-component* shall be a variable.

C560 (R528) The *scalar-structure-component* shall contain at least one *part-ref* that contains a *subscript-list*.

C561 (R528) In an *array-element* or a *scalar-structure-component* that is a *data-i-do-object*, any subscript shall be an expression whose primaries are either constants, subobjects of constants, or DO variables of this *data-implied-do* or the containing *data-implied-do*s, and each operation shall be intrinsic.

R530	*data-stmt-value*	**is**	[*data-stmt-repeat* *] *data-stmt-constant*
R531	*data-stmt-repeat*	**is**	*scalar-int-constant*
		or	*scalar-int-constant-subobject*

C562 (R531) The *data-stmt-repeat* shall be positive or zero. If the *data-stmt-repeat* is a named constant, it shall have been declared previously in the scoping unit or made accessible by use association or host association.

R532	*data-stmt-constant*	**is**	*scalar-constant*
		or	*scalar-constant-subobject*
		or	*signed-int-literal-constant*
		or	*signed-real-literal-constant*
		or	*null-init*
		or	*structure-constructor*

C563 (R532) If a DATA statement constant value is a named constant or a structure constructor, the named constant or derived type shall have been declared previously in the scoping unit or made accessible by use or host association.

C564 (R532) If a *data-stmt-constant* is a *structure-constructor*, it shall be an initialization expression.

R533	*int-constant-subobject*	**is**	*constant-subobject*

C565 (R533) *int-constant-subobject* shall be of type integer.

R534	*constant-subobject*	**is**	*designator*

C566 (R534) *constant-subobject* shall be a subobject of a constant.

C567 (R534) Any subscript, substring starting point, or substring ending point shall be an initialization expression.

The *data-stmt-object-list* is expanded to form a sequence of pointers and scalar variables, referred to as "sequence of variables" in subsequent text. A nonpointer array whose unqualified name appears in a *data-stmt-object-list* is equivalent to a complete sequence of its array elements in array element order

(6.2.2.2). An array section is equivalent to the sequence of its array elements in array element order. A *data-implied-do* is expanded to form a sequence of array elements and structure components, under the control of the *data-i-do-variable*, as in the DO construct (8.1.6.4).

The *data-stmt-value-list* is expanded to form a sequence of *data-stmt-constant*s. A *data-stmt-repeat* indicates the number of times the following *data-stmt-constant* is to be included in the sequence; omission of a *data-stmt-repeat* has the effect of a repeat factor of 1.

A zero-sized array or a *data-implied-do* with an iteration count of zero contributes no variables to the expanded sequence of variables, but a zero-length scalar character variable does contribute a variable to the expanded sequence. A *data-stmt-constant* with a repeat factor of zero contributes no *data-stmt-constant*s to the expanded sequence of scalar *data-stmt-constant*s.

The expanded sequences of variables and *data-stmt-constant*s are in one-to-one correspondence. Each *data-stmt-constant* specifies the initial value or *null-init* for the corresponding variable. The lengths of the two expanded sequences shall be the same.

A *data-stmt-constant* shall be *null-init* if and only if the corresponding *data-stmt-object* has the POINTER attribute. The initial association status of a pointer *data-stmt-object* is disassociated.

A *data-stmt-constant* other than *null-init* shall be compatible with its corresponding variable according to the rules of intrinsic assignment (7.4.1.2). The variable is initially defined with the value specified by the *data-stmt-constant*; if necessary, the value is converted according to the rules of intrinsic assignment (7.4.1.3) to a value that agrees in type, type parameters, and shape with the variable.

If a *data-stmt-constant* is a *boz-literal-constant*, the corresponding variable shall be of type integer. The *boz-literal-constant* is treated as if it were an *int-literal-constant* with a *kind-param* that specifies the representation method with the largest decimal exponent range supported by the processor.

NOTE 5.25

Examples of DATA statements are:

```
CHARACTER (LEN = 10)  NAME
INTEGER, DIMENSION (0:9) :: MILES
REAL, DIMENSION (100, 100) :: SKEW
TYPE (NODE), POINTER :: HEAD_OF_LIST
TYPE (PERSON) MYNAME, YOURNAME
DATA NAME / 'JOHN DOE' /, MILES / 10 * 0 /
DATA ((SKEW (K, J), J = 1, K), K = 1, 100) / 5050 * 0.0 /
DATA ((SKEW (K, J), J = K + 1, 100), K = 1, 99) / 4950 * 1.0 /
DATA HEAD_OF_LIST / NULL() /
DATA MYNAME / PERSON (21, 'JOHN SMITH') /
DATA YOURNAME % AGE, YOURNAME % NAME / 35, 'FRED BROWN' /
```

The character variable NAME is initialized with the value JOHN DOE with padding on the right because the length of the constant is less than the length of the variable. All ten elements of the integer array MILES are initialized to zero. The two-dimensional array SKEW is initialized so that the lower triangle of SKEW is zero and the strict upper triangle is one. The structures MYNAME and YOURNAME are declared using the derived type PERSON from Note 4.17. The pointer HEAD_OF_LIST is declared using the derived type NODE from Note 4.36; it is initially disassociated. MYNAME is initialized by a structure constructor. YOURNAME is initialized by supplying a separate value for each component.

5.2.6 DIMENSION statement

R535 *dimension-stmt* **is** DIMENSION [::] *array-name* (*array-spec*) [, *array-name* (*array-spec*)] ...

This statement specifies the DIMENSION attribute (5.1.2.5) and the array properties for each object named.

NOTE 5.26

An example of a DIMENSION statement is:

```
DIMENSION A (10), B (10, 70), C (:)
```

5.2.7 INTENT statement

R536 *intent-stmt* **is** INTENT (*intent-spec*) [::] *dummy-arg-name-list*

This statement specifies the INTENT attribute (5.1.2.7) for the dummy arguments in the list.

NOTE 5.27

An example of an INTENT statement is:

```
SUBROUTINE EX (A, B)
   INTENT (INOUT) :: A, B
```

5.2.8 OPTIONAL statement

R537 *optional-stmt* **is** OPTIONAL [::] *dummy-arg-name-list*

This statement specifies the OPTIONAL attribute (5.1.2.9) for the dummy arguments in the list.

NOTE 5.28

An example of an OPTIONAL statement is:

```
SUBROUTINE EX (A, B)
   OPTIONAL :: B
```

5.2.9 PARAMETER statement

The **PARAMETER statement** specifies the PARAMETER attribute (5.1.2.10) and the values for the named constants in the list.

R538 *parameter-stmt* **is** PARAMETER (*named-constant-def-list*)
R539 *named-constant-def* **is** *named-constant* = *initialization-expr*

The named constant shall have its type, type parameters, and shape specified in a prior specification of the *specification-part* or declared implicitly (5.3). If the named constant is typed by the implicit typing rules, its appearance in any subsequent specification of the *specification-part* shall confirm this implied type and the values of any implied type parameters.

The value of each named constant is that specified by the corresponding initialization expression; if necessary, the value is converted according to the rules of intrinsic assignment (7.4.1.3) to a value that agrees in type, type parameters, and shape with the named constant.

NOTE 5.29

An example of a PARAMETER statement is:

```
PARAMETER (MODULUS = MOD (28, 3), NUMBER_OF_SENATORS = 100)
```

5.2.10 POINTER statement

R540	*pointer-stmt*	**is**	POINTER [::] *pointer-decl-list*
R541	*pointer-decl*	**is**	*object-name* [(*deferred-shape-spec-list*)]
		or	*proc-entity-name*

C568 (R541) A *proc-entity-name* shall also be declared in a *procedure-declaration-stmt*.

This statement specifies the POINTER attribute (5.1.2.11) for a list of objects and procedure entities.

NOTE 5.30

An example of a POINTER statement is:

```
TYPE (NODE) :: CURRENT
POINTER :: CURRENT, A (:, :)
```

5.2.11 PROTECTED statement

R542	*protected-stmt*	**is**	PROTECTED [::] *entity-name-list*

The **PROTECTED statement** specifies the PROTECTED attribute (5.1.2.12) for a list of entities.

5.2.12 SAVE statement

R543	*save-stmt*	**is**	SAVE [[::] *saved-entity-list*]
R544	*saved-entity*	**is**	*object-name*
		or	*proc-pointer-name*
		or	/ *common-block-name* /
R545	*proc-pointer-name*	**is**	*name*

C569 (R545) A *proc-pointer-name* shall be the name of a procedure pointer.

C570 (R543) If a SAVE statement with an omitted saved entity list occurs in a scoping unit, no other explicit occurrence of the SAVE attribute or SAVE statement is permitted in the same scoping unit.

A SAVE statement with a saved entity list specifies the SAVE attribute (5.1.2.13) for all entities named in the list or included within a common block named in the list. A SAVE statement without a saved entity list is treated as though it contained the names of all allowed items in the same scoping unit.

If a particular common block name is specified in a SAVE statement in any scoping unit of a program other than the main program, it shall be specified in a SAVE statement in every scoping unit in which that common block appears except in the scoping unit of the main program. For a common block declared in a SAVE statement, the values in the common block storage sequence (5.5.2.1) at the time a RETURN or END statement is executed are made available to the next scoping unit in the execution sequence of the program that specifies the common block name or accesses the common block. If a named common block is specified in the scoping unit of the main program, the current values of the common block storage sequence are made available to each scoping unit that specifies the named common block. 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 may appear in the specification part of a main program and has no effect.

NOTE 5.31

An example of a SAVE statement is:

```
SAVE A, B, C, / BLOCKA /, D
```

5.2.13 TARGET statement

R546	*target-stmt*	**is**	TARGET [::] *object-name* [(*array-spec*)] ■ ■ [, *object-name* [(*array-spec*)]] ...

This statement specifies the TARGET attribute (5.1.2.14) for a list of objects.

NOTE 5.32

An example of a TARGET statement is:

```
TARGET :: A (1000, 1000), B
```

5.2.14 VALUE statement

R547	*value-stmt*	**is**	VALUE [::] *dummy-arg-name-list*

The VALUE statement specifies the VALUE attribute (5.1.2.15) for a list of dummy arguments.

5.2.15 VOLATILE statement

R548	*volatile-stmt*	**is**	VOLATILE [::] *object-name-list*

The VOLATILE statement specifies the VOLATILE attribute (5.1.2.16) for a list of objects.

5.3 IMPLICIT statement

In a scoping unit, an **IMPLICIT statement** specifies a type, and possibly type parameters, for all implicitly typed data entities whose names begin with one of the letters specified in the statement. Alternatively, it may indicate that no implicit typing rules are to apply in a particular scoping unit.

R549	*implicit-stmt*	**is**	IMPLICIT *implicit-spec-list*
		or	IMPLICIT NONE
R550	*implicit-spec*	**is**	*declaration-type-spec* (*letter-spec-list*)
R551	*letter-spec*	**is**	*letter* [– *letter*]

C571 (R549) If IMPLICIT NONE is specified in a scoping unit, it shall precede any PARAMETER statements that appear in the scoping unit and there shall be no other IMPLICIT statements in the scoping unit.

C572 (R551) If the minus and second letter appear, the second letter shall follow the first letter alphabetically.

A *letter-spec* consisting of two letters separated by a minus is equivalent to writing a list containing 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. The same letter shall 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 scoping unit.

In each scoping unit, there is a mapping, which may be null, between each of the letters A, B, ..., Z and a type (and type parameters). An IMPLICIT statement specifies the mapping for the letters in its *letter-spec-list*. IMPLICIT NONE specifies the null mapping for all the letters. If a mapping is not specified for a letter, the default for a program unit or an interface body is default integer if the letter is I, J, ..., or N and default real otherwise, and the default for an internal or module procedure is the mapping in the host scoping unit.

Any data entity that is not explicitly declared by a type declaration statement, is not an intrinsic function, and is not made accessible by use association or host association is declared implicitly to be of the type (and type parameters) mapped from the first letter of its name, provided the mapping is not null. The mapping for the first letter of the data entity shall either have been established by a prior IMPLICIT statement or be the default mapping for the letter. The mapping may be to a derived type that is inaccessible in the local scope if the derived type is accessible to the host scope. The data entity is treated as if it were declared in an explicit type declaration in the outermost scoping unit in which it appears. An explicit type specification in a FUNCTION statement overrides an IMPLICIT statement for the name of the result variable of that function subprogram.

NOTE 5.33

The following are examples of the use of IMPLICIT statements:

```
MODULE EXAMPLE_MODULE
   IMPLICIT NONE
   ...
   INTERFACE
      FUNCTION FUN (I)   ! Not all data entities need to
         INTEGER FUN     ! be declared explicitly
      END FUNCTION FUN
   END INTERFACE
CONTAINS
   FUNCTION JFUN (J)     ! All data entities need to
      INTEGER JFUN, J    ! be declared explicitly.
      ...
   END FUNCTION JFUN
END MODULE EXAMPLE_MODULE
SUBROUTINE SUB
   IMPLICIT COMPLEX (C)
   C = (3.0, 2.0)     ! C is implicitly declared COMPLEX
   ...
CONTAINS
   SUBROUTINE SUB1
      IMPLICIT INTEGER (A, C)
      C = (0.0, 0.0)  ! C is host associated and of
                      ! type complex
      Z = 1.0         ! Z is implicitly declared REAL
      A = 2           ! A is implicitly declared INTEGER
      CC = 1          ! CC is implicitly declared INTEGER
      ...
   END SUBROUTINE SUB1
   SUBROUTINE SUB2
      Z = 2.0         ! Z is implicitly declared REAL and
                      ! is different from the variable of
                      ! the same name in SUB1
```

NOTE 5.33 (cont.)

```
      ...
   END SUBROUTINE SUB2
   SUBROUTINE SUB3
      USE EXAMPLE_MODULE  ! Accesses integer function FUN
                          ! by use association
      Q = FUN (K)         ! Q is implicitly declared REAL and
      ...                 ! K is implicitly declared INTEGER
   END SUBROUTINE SUB3
END SUBROUTINE SUB
```

NOTE 5.34

An IMPLICIT statement may specify a *declaration-type-spec* of derived type.

For example, given an IMPLICIT statement and a type defined as follows:

```
IMPLICIT TYPE (POSN) (A-B, W-Z), INTEGER (C-V)
TYPE POSN
   REAL X, Y
   INTEGER Z
END TYPE POSN
```

variables beginning with the letters A, B, W, X, Y, and Z are implicitly typed with the type POSN and the remaining variables are implicitly typed with type INTEGER.

NOTE 5.35

The following is an example of a mapping to a derived type that is inaccessible in the local scope:

```
     PROGRAM MAIN
       IMPLICIT TYPE(BLOB) (A)
       TYPE BLOB
         INTEGER :: I
       END TYPE BLOB
       TYPE(BLOB) :: B
       CALL STEVE
     CONTAINS
       SUBROUTINE STEVE
         INTEGER :: BLOB
         ..
         AA = B
         ..
       END SUBROUTINE STEVE
     END PROGRAM MAIN
```

In the subroutine STEVE, it is not possible to explicitly declare a variable to be of type BLOB because BLOB has been given a different meaning, but implicit mapping for the letter A still maps to type BLOB, so AA is of type BLOB.

5.4 NAMELIST statement

A **NAMELIST statement** specifies a group of named data objects, which may be referred to by a single name for the purpose of data transfer (9.5, 10.10).

R552 *namelist-stmt* **is** NAMELIST ■
■ / *namelist-group-name* / *namelist-group-object-list* ■
■ [[,] / *namelist-group-name* / ■
■ *namelist-group-object-list*] . . .

C573 (R552) The *namelist-group-name* shall not be a name made accessible by use association.

R553 *namelist-group-object* **is** *variable-name*

C574 (R553) A *namelist-group-object* shall not be an assumed-size array.

C575 (R552) A *namelist-group-object* shall not have the PRIVATE attribute if the *namelist-group-name* has the PUBLIC attribute.

The order in which the variables are specified in the NAMELIST statement determines the order in which the values appear on output.

Any *namelist-group-name* may occur more than once in the NAMELIST statements in a scoping unit. The *namelist-group-object-list* following each successive appearance of the same *namelist-group-name* in a scoping unit is treated as a continuation of the list for that *namelist-group-name*.

A namelist group object may be a member of more than one namelist group.

A namelist group object shall either be accessed by use or host association or shall have its type, type parameters, and shape specified by previous specification statements or the procedure heading in the same scoping unit or by the implicit typing rules in effect for the scoping unit. If a namelist group object is typed by the implicit typing rules, its appearance in any subsequent type declaration statement shall confirm the implied type and type parameters.

NOTE 5.36

An example of a NAMELIST statement is:

```
NAMELIST /NLIST/ A, B, C
```

5.5 Storage association of data objects

In general, the physical storage units or storage order for data objects is not specifiable. However, the EQUIVALENCE, COMMON, and SEQUENCE statements and the BIND(C) *type-attr-spec* provide for control of the order and layout of storage units. The general mechanism of storage association is described in 16.4.3.

5.5.1 EQUIVALENCE statement

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

If the equivalenced objects have differing type or type parameters, the EQUIVALENCE statement does not cause type conversion or imply mathematical equivalence. 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.

R554	*equivalence-stmt*	**is**	EQUIVALENCE *equivalence-set-list*
R555	*equivalence-set*	**is**	(*equivalence-object* , *equivalence-object-list*)
R556	*equivalence-object*	**is**	*variable-name*
		or	*array-element*
		or	*substring*

C576 (R556) An *equivalence-object* shall not be a designator with a base object that is a dummy argument, a pointer, an allocatable variable, a derived-type object that has an allocatable ultimate component, an object of a nonsequence derived type, an object of a derived type that has a pointer at any level of component selection, an automatic object, a function name, an entry name, a result name, a variable with the BIND attribute, a variable in a common block that has the BIND attribute, or a named constant.

C577 (R556) An *equivalence-object* shall not be a designator that has more than one *part-ref*.

C578 (R556) An *equivalence-object* shall not have the TARGET attribute.

C579 (R556) Each subscript or substring range expression in an *equivalence-object* shall be an integer initialization expression (7.1.7).

C580 (R555) If an *equivalence-object* is of type default integer, default real, double precision real, default complex, default logical, or numeric sequence type, all of the objects in the equivalence set shall be of these types.

C581 (R555) If an *equivalence-object* is of type default character or character sequence type, all of the objects in the equivalence set shall be of these types.

C582 (R555) If an *equivalence-object* is of a sequence derived type that is not a numeric sequence or character sequence type, all of the objects in the equivalence set shall be of the same type with the same type parameter values.

C583 (R555) If an *equivalence-object* is of an intrinsic type other than default integer, default real, double precision real, default complex, default logical, or default character, all of the objects in the equivalence set shall be of the same type with the same kind type parameter value.

C584 (R556) If an *equivalence-object* has the PROTECTED attribute, all of the objects in the equivalence set shall have the PROTECTED attribute.

C585 (R556) The name of an *equivalence-object* shall not be a name made accessible by use association.

C586 (R556) A *substring* shall not have length zero.

NOTE 5.37

The EQUIVALENCE statement allows the equivalencing of sequence structures and the equivalencing of objects of intrinsic type with nondefault type parameters, but there are strict rules regarding the appearance of these objects in an EQUIVALENCE statement.

A structure that appears in an EQUIVALENCE statement shall be a sequence structure. If a sequence structure is not of numeric sequence type or of character sequence type, it shall be equivalenced only to objects of the same type with the same type parameter values.

A structure of a numeric sequence type may be equivalenced to another structure of a numeric sequence type, an object of default integer type, default real type, double precision real type, default complex type, or default logical type such that components of the structure ultimately become associated only with objects of these types.

A structure of a character sequence type may be equivalenced to an object of default character

NOTE 5.37 (cont.)

type or another structure of a character sequence type.

An object of intrinsic type with nondefault kind type parameters may be equivalenced only to objects of the same type and kind type parameters.

Further rules on the interaction of EQUIVALENCE statements and default initialization are given in 16.4.3.3.

5.5.1.1 Equivalence association

An EQUIVALENCE statement specifies that the storage sequences (16.4.3.1) of the data objects specified in an *equivalence-set* are storage associated. All of the nonzero-sized sequences in the *equivalence-set*, if any, have the same first storage unit, and all of the zero-sized sequences in the *equivalence-set*, if any, are storage associated with one another and with the first storage unit of any nonzero-sized sequences. This causes the storage association of the data objects in the *equivalence-set* and may cause storage association of other data objects.

5.5.1.2 Equivalence of default character objects

A data object of type default character may be equivalenced only with other objects of type default character. The lengths of the equivalenced objects need not be the same.

An EQUIVALENCE statement specifies that the storage sequences of all the default character data objects specified in an *equivalence-set* are storage associated. All of the nonzero-sized sequences in the *equivalence-set*, if any, have the same first character storage unit, and all of the zero-sized sequences in the *equivalence-set*, if any, are storage associated with one another and with the first character storage unit of any nonzero-sized sequences. This causes the storage association of the data objects in the *equivalence-set* and may cause storage association of other data objects.

NOTE 5.38

For example, using the declarations:

```
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)   ---|
```

5.5.1.3 Array names and array element designators

For a nonzero-sized array, the use of the array name unqualified by a subscript list in an EQUIVALENCE statement has the same effect as using an array element designator that identifies the first element of the array.

5.5.1.4 Restrictions on EQUIVALENCE statements

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

NOTE 5.39

For example:

```
REAL, DIMENSION (2) :: A
REAL :: B
EQUIVALENCE (A (1), B), (A (2), B) ! Not standard conforming
```

is prohibited, because it would specify the same storage unit for A (1) and A (2).

An EQUIVALENCE statement shall not specify that consecutive storage units are to be nonconsecutive.

NOTE 5.40

For example, the following is prohibited:

```
REAL A (2)
DOUBLE PRECISION D (2)
EQUIVALENCE (A (1), D (1)), (A (2), D (2)) ! Not standard conforming
```

5.5.2 COMMON statement

The **COMMON statement** specifies blocks of physical storage, called **common blocks**, that may be accessed by any of the scoping units in a program. Thus, the COMMON statement provides a global data facility based on storage association (16.4.3).

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**.

R557	*common-stmt*	**is**	COMMON ■
			■ [/ [*common-block-name*] /] *common-block-object-list* ■
			■ [[,] / [*common-block-name*] / ■
			■ *common-block-object-list*] ...
R558	*common-block-object*	**is**	*variable-name* [(*explicit-shape-spec-list*)]
		or	*proc-pointer-name*

C587 (R558) Only one appearance of a given *variable-name* or *proc-pointer-name* is permitted in all *common-block-object-list*s within a scoping unit.

C588 (R558) A *common-block-object* shall not be a dummy argument, an allocatable variable, a derived-type object with an ultimate component that is allocatable, an automatic object, a function name, an entry name, a variable with the BIND attribute, or a result name.

C589 (R558) If a *common-block-object* is of a derived type, it shall be a sequence type (4.5.1) or a type with the BIND attribute and it shall have no default initialization.

C590 (R558) A *variable-name* or *proc-pointer-name* shall not be a name made accessible by use association.

In each COMMON statement, the data objects whose names appear in a common block object 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 object 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 object 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 scoping unit. The common block list following each successive appearance of the same common block name in a scoping unit is treated as a continuation of the list for that common block name. Similarly, each blank common block object list in a scoping unit is treated as a continuation of blank common.

The form *variable-name* (*explicit-shape-spec-list*) declares *variable-name* to have the DIMENSION attribute and specifies the array properties that apply. If derived-type objects of numeric sequence type (4.5.1) or character sequence type (4.5.1) appear in common, it is as if the individual components were enumerated directly in the common list.

NOTE 5.41

Examples of COMMON statements are:

```
COMMON /BLOCKA/ A, B, D (10, 30)
COMMON I, J, K
```

5.5.2.1 Common block storage sequence

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

(1) A storage sequence is formed consisting of the sequence of storage units in the storage sequences (16.4.3.1) of all data objects in the common block object lists for the common block. The order of the storage sequences is the same as the order of the appearance of the common block object lists in the scoping 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.

Only COMMON statements and EQUIVALENCE statements appearing in the scoping unit contribute to common block storage sequences formed in that unit.

5.5.2.2 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.

5.5.2.3 Common association

Within a program, the common block storage sequences of all nonzero-sized common blocks with the same name have the same first storage unit, and the common block storage sequences of all zero-sized common blocks with the same name are storage associated with one another. Within a program, the common block storage sequences of all nonzero-sized blank common blocks have the same first storage unit and the storage sequences of all zero-sized blank common blocks are associated with one another and with the first storage unit of any nonzero-sized blank common blocks. This results in the association of objects in different scoping units. Use association or host association may cause these associated objects to be accessible in the same scoping unit.

A nonpointer object of default integer type, default real type, double precision real type, default complex type, default logical type, or numeric sequence type shall be associated only with nonpointer objects of these types.

A nonpointer object of type default character or character sequence type shall be associated only with nonpointer objects of these types.

A nonpointer object of a derived type that is not a numeric sequence or character sequence type shall be associated only with nonpointer objects of the same type with the same type parameter values.

A nonpointer object of intrinsic type other than default integer, default real, double precision real, default complex, default logical, or default character shall be associated only with nonpointer objects of the same type and type parameters.

A data pointer shall be storage associated only with data pointers of the same type and rank. Data pointers that are storage associated shall have deferred the same type parameters; corresponding nondeferred type parameters shall have the same value. A procedure pointer shall be storage associated only with another procedure pointer; either both interfaces shall be explicit or both interfaces shall be implicit. If the interfaces are explicit, the characteristics shall be the same. If the interfaces are implicit, either both shall be subroutines or both shall be functions with the same type and type parameters.

An object with the TARGET attribute may be storage associated only with another object that has the TARGET attribute and the same type and type parameters.

NOTE 5.42

A common block is permitted to contain sequences of different storage units, provided each scoping unit that accesses the common block specifies an identical sequence of storage units for the common block. For example, this allows a single common block to contain both numeric and character storage units.

Association in different scoping units between objects of default type, objects of double precision real type, and sequence structures is permitted according to the rules for equivalence objects (5.5.1).

5.5.2.4 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 may cause data objects in a named common block 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 (16.5.6).

(2) Named common blocks of the same name shall be of the same size in all scoping units of a 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 or type declaration statement in a block data program unit (11.3), but objects in blank common shall not be initially defined.

5.5.2.5 Restrictions on common and equivalence

An EQUIVALENCE statement shall not cause the storage sequences of two different common blocks to be associated.

Equivalence association shall 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.

NOTE 5.43

For example, the following is not permitted:

```
COMMON /X/ A
REAL B (2)
EQUIVALENCE (A, B (2))    ! Not standard conforming
```

Equivalence association shall not cause a derived-type object with default initialization to be associated with an object in a common block.

Section 6: Use of data objects

The appearance of a data object designator in a context that requires its value is termed a reference. A reference is permitted only if the data object is defined. A reference to a pointer is permitted only if the pointer is associated with a target object that is defined. A data object becomes defined with a value when events described in 16.5.5 occur.

R601 *variable* **is** *designator*

C601 (R601) *designator* shall not be a constant or a subobject of a constant.

R602 *variable-name* **is** *name*

C602 (R602) A *variable-name* shall be the name of a variable.

R603 *designator* **is** *object-name*
or *array-element*
or *array-section*
or *structure-component*
or *substring*

R604 *logical-variable* **is** *variable*

C603 (R604) *logical-variable* shall be of type logical.

R605 *default-logical-variable* **is** *variable*

C604 (R605) *default-logical-variable* shall be of type default logical.

R606 *char-variable* **is** *variable*

C605 (R606) *char-variable* shall be of type character.

R607 *default-char-variable* **is** *variable*

C606 (R607) *default-char-variable* shall be of type default character.

R608 *int-variable* **is** *variable*

C607 (R608) *int-variable* shall be of type integer.

NOTE 6.1

For example, given the declarations:

```
CHARACTER (10)  A, B (10)
TYPE (PERSON)  P   ! See Note 4.17
```

then A, B, B (1), B (1:5), P % AGE, and A (1:1) are all variables.

A constant (3.2.2) is a literal constant or a named constant. A literal constant is a scalar denoted by a syntactic form, which indicates its type, type parameters, and value. A named constant is a constant that has a name; the name has the PARAMETER attribute (5.1.2.10, 5.2.9). A reference to a constant is always permitted; redefinition of a constant is never permitted.

6.1 Scalars

A **scalar** (2.4.4) is a data entity that can be represented by a single value of the type and that is not an array (6.2). Its value, if defined, is a single element from the set of values that characterize its type.

NOTE 6.2

A scalar object of derived type has a single value that consists of the values of its components (4.5.7).

A scalar has rank zero.

6.1.1 Substrings

A **substring** is a contiguous portion of a character string (4.4.4). The following rules define the forms of a substring:

R609	*substring*	**is**	*parent-string* (*substring-range*)
R610	*parent-string*	**is**	*scalar-variable-name*
		or	*array-element*
		or	*scalar-structure-component*
		or	*scalar-constant*
R611	*substring-range*	**is**	[*scalar-int-expr*] : [*scalar-int-expr*]

C608 (R610) *parent-string* shall be of type character.

The value of the first *scalar-int-expr* in *substring-range* is called the **starting point** and the value of the second one is called the **ending point**. The length of a substring is the number of characters in the substring and is MAX $(l - f + 1, 0)$, where f and l are the starting and ending points, respectively.

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 shall be within the range 1, 2, ..., n unless the starting point exceeds the ending point, in which case the substring has length zero. If the starting point is not specified, the default value is 1. If the ending point is not specified, the default value is n.

If the parent is a variable, the substring is also a variable.

NOTE 6.3

Examples of character substrings are:

```
B(1)(1:5)           array element as parent string
P%NAME(1:1)         structure component as parent string
ID(4:9)             scalar variable name as parent string
'0123456789'(N:N)   character constant as parent string
```

6.1.2 Structure components

A **structure component** is part of an object of derived type; it may be referenced by an object designator. A structure component may be a scalar or an array.

R612 *data-ref* **is** *part-ref* [% *part-ref*] ...
R613 *part-ref* **is** *part-name* [(*section-subscript-list*)]

C609 (R612) Each *part-name* except the rightmost shall be of derived type.

C610 (R612) Each *part-name* except the leftmost shall be the name of a component of the declared type of the preceding *part-name*.

C611 (R612) If the rightmost *part-name* is of abstract type, *data-ref* shall be polymorphic.

C612 (R612) The leftmost *part-name* shall be the name of a data object.

C613 (R613) If a *section-subscript-list* appears, the number of *section-subscript*s shall equal the rank of *part-name*.

The rank of a *part-ref* of the form *part-name* is the rank of *part-name*. The rank of a *part-ref* that has a section subscript list is the number of subscript triplets and vector subscripts in the list.

C614 (R612) There shall not be more than one *part-ref* with nonzero rank. A *part-name* to the right of a *part-ref* with nonzero rank shall not have the ALLOCATABLE or POINTER attribute.

The rank of a *data-ref* is the rank of the *part-ref* with nonzero rank, if any; otherwise, the rank is zero. The **base object** of a *data-ref* is the data object whose name is the leftmost part name.

The type and type parameters, if any, of a *data-ref* are those of the rightmost part name.

A *data-ref* with more than one *part-ref* is a subobject of its base object if none of the *part-name*s, except for possibly the rightmost, are pointers. If the rightmost *part-name* is the only pointer, then the *data-ref* is a subobject of its base object in contexts that pertain to its pointer association status but not in any other contexts.

NOTE 6.4

If X is an object of derived type with a pointer component P, then the pointer X%P is a subobject of X when considered as a pointer – that is in contexts where it is not dereferenced.

However the target of X%P is not a subobject of X. Thus, in contexts where X%P is dereferenced to refer to the target, it is not a subobject of X.

R614 *structure-component* **is** *data-ref*

C615 (R614) There shall be more than one *part-ref* and the rightmost *part-ref* shall be of the form *part-name*.

A structure component shall be neither referenced nor defined before the declaration of the base object. A structure component is a pointer only if the rightmost part name is defined to have the POINTER attribute.

NOTE 6.5

Examples of structure components are:

```
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
```

For a more elaborate example see C.3.1.

NOTE 6.6

The syntax rules are structured such that a *data-ref* that ends in a component name without a following subscript list is a structure component, even when other component names in the *data-ref* are followed by a subscript list. A *data-ref* that ends in a component name with a following subscript list is either an array element or an array section. A *data-ref* of nonzero rank that ends with a *substring-range* is an array section. A *data-ref* of zero rank that ends with a *substring-range* is a substring.

A **subcomponent** of an object of derived type is a component of that object or of a subobject of that object.

6.1.3 Type parameter inquiry

A **type parameter inquiry** is used to inquire about a type parameter of a data object. It applies to both intrinsic and derived types.

R615 *type-param-inquiry* **is** *designator* % *type-param-name*

C616 (R615) The *type-param-name* shall be the name of a type parameter of the declared type of the object designated by the *designator*.

A deferred type parameter of a pointer that is not associated or of an unallocated allocatable variable shall not be inquired about.

NOTE 6.7

A *type-param-inquiry* has a syntax like that of a structure component reference, but it does not have the same semantics. It is not a variable and thus can never be assigned to. It may be used only as a primary in an expression. It is scalar even if *designator* is an array.

The intrinsic type parameters can also be inquired about by using the intrinsic functions KIND and LEN.

NOTE 6.8

The following are examples of type parameter inquiries:

```
a%kind      !-- A is real.  Same value as KIND(a).
s%len       !-- S is character.  Same value as LEN(s).
b(10)%kind  !-- Inquiry about an array element.
p%dim       !-- P is of the derived type general_point.
```

See Note 4.24 for the definition of the `general_point` type used in the last example above.

6.2 Arrays

An **array** is a set of scalar data, all of the same type and type parameters, whose individual elements are arranged in a rectangular pattern. The scalar data that make up an array are the **array elements**.

No order of reference to the elements of an array is indicated by the appearance of the array designator, except where array element ordering (6.2.2.2) is specified.

6.2.1 Whole arrays

A **whole array** is a named array, which may be either a named constant (5.1.2.10, 5.2.9) or a variable; no subscript list is appended to the name.

The appearance of a whole array variable in an executable construct specifies all the elements of the array (2.4.5). An assumed-size array is permitted to appear as a whole array in an executable construct only as an actual argument in a procedure reference that does not require the shape.

The appearance of a whole array name in a nonexecutable statement specifies the entire array except for the appearance of a whole array name in an equivalence set (5.5.1.3).

6.2.2 Array elements and array sections

R616	*array-element*	**is**	*data-ref*

C617 (R616) Every *part-ref* shall have rank zero and the last *part-ref* shall contain a *subscript-list*.

R617	*array-section*	**is**	*data-ref* [(*substring-range*)]

C618 (R617) Exactly one *part-ref* shall have nonzero rank, and either the final *part-ref* shall have a *section-subscript-list* with nonzero rank or another *part-ref* shall have nonzero rank.

C619 (R617) If a *substring-range* appears, the rightmost *part-name* shall be of type character.

R618	*subscript*	**is**	*scalar-int-expr*
R619	*section-subscript*	**is**	*subscript*
		or	*subscript-triplet*
		or	*vector-subscript*
R620	*subscript-triplet*	**is**	[*subscript*] : [*subscript*] [: *stride*]
R621	*stride*	**is**	*scalar-int-expr*
R622	*vector-subscript*	**is**	*int-expr*

C620 (R622) A *vector-subscript* shall be an integer array expression of rank one.

C621 (R620) The second subscript shall not be omitted from a *subscript-triplet* in the last dimension of an assumed-size array.

An array element is a scalar. An array section is an array. If a *substring-range* is present in an *array-section*, each element is the designated substring of the corresponding element of the array section.

NOTE 6.9

For example, with the declarations:

```
REAL A (10, 10)
CHARACTER (LEN = 10) B (5, 5, 5)
```

A (1, 2) is an array element, A (1:N:2, M) is a rank-one array section, and B (:, :, :) (2:3) is an array of shape (5, 5, 5) whose elements are substrings of length 2 of the corresponding elements of B.

NOTE 6.10

Unless otherwise specified, an array element or array section does not have an attribute of the whole array. In particular, an array element or an array section does not have the POINTER or ALLOCATABLE attribute.

NOTE 6.11

Examples of array elements and array sections are:

```
ARRAY_A(1:N:2)%ARRAY_B(I, J)%STRING(K)(:)     array section
SCALAR_PARENT%ARRAY_FIELD(J)                  array element
SCALAR_PARENT%ARRAY_FIELD(1:N)                array section
SCALAR_PARENT%ARRAY_FIELD(1:N)%SCALAR_FIELD   array section
```

6.2.2.1 Array elements

The value of a subscript in an array element shall be within the bounds for that dimension.

6.2.2.2 Array element order

The elements of an array form a sequence known as the **array element order**. 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	Subscript bounds	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	$1 + (s_1 - j_1)$ $+ (s_2 - j_2) \times d_1$ $+ (s_3 - j_3) \times d_2 \times d_1$
. . .	. . .	. . .	. . .
7	$j_1{:}k_1, \ldots, j_7{:}k_7$	$s_1, \ldots, s_7$	$1 + (s_1 - j_1)$ $+ (s_2 - j_2) \times d_1$ $+ (s_3 - j_3) \times d_2 \times d_1$ $+ \ldots$ $+ (s_7 - j_7) \times d_6$ $\times d_5 \times \ldots \times d_1$

Notes for Table 6.1:

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

2) If the size of the array is nonzero, $j_i \le s_i \le k_i$ for all $i = 1, 2, ..., 7$.

6.2.2.3 Array sections

An **array section** is an array subobject optionally followed by a substring range.

In an *array-section* having a *section-subscript-list*, each *subscript-triplet* and *vector-subscript* in the section subscript list indicates a sequence of subscripts, which may be empty. Each subscript in such a sequence shall be within the bounds for its dimension unless the sequence is empty. The array section is the set of elements from the array determined by all possible subscript lists obtainable from the single subscripts or sequences of subscripts specified by each section subscript.

In an *array-section* with no *section-subscript-list*, the rank and shape of the array is the rank and shape of the *part-ref* with nonzero rank; otherwise, the rank of the array section is the number of subscript triplets and vector subscripts 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 subscript triplet or vector subscript. 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.

6.2.2.3.1 Subscript triplet

A subscript triplet designates a regular sequence of subscripts consisting of zero or more subscript values. The third expression in the subscript triplet is the increment between the subscript values and is called the **stride**. The subscripts and stride of a subscript triplet are optional. An omitted first subscript in a subscript triplet is equivalent to a subscript whose value is the lower bound for the array and an omitted second subscript is equivalent to the upper bound. An omitted stride is equivalent to a stride of 1.

The stride shall not be zero.

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 greater than the second subscript; the sequence is empty if the first subscript is greater than the second.

NOTE 6.12

For 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):

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

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 greater than the second subscript; the sequence is empty if the second subscript is greater than the first.

NOTE 6.13

For example, if an array is declared B (10), 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.

NOTE 6.14

A subscript in a subscript triplet need not be within the declared bounds for that dimension if all values used in selecting the array elements are within the declared bounds.

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.

6.2.2.3.2 Vector subscript

A **vector subscript** designates a sequence of subscripts corresponding to the values of the elements of the expression. Each element of the expression shall be defined. A **many-one array section** is an array section with a vector subscript having two or more elements with the same value. A many-one array section shall appear neither on the left of the equals in an assignment statement nor as an input item in a READ statement.

An array section with a vector subscript shall not be argument associated with a dummy array that is defined or redefined. An array section with a vector subscript shall not be the target in a pointer assignment statement. An array section with a vector subscript shall not be an internal file.

NOTE 6.15

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 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 (3, V) and Z (U, V) contain duplicate elements from Z, the sections Z (3, V) and Z (U, V) shall not be redefined as sections.

6.3 Dynamic association

Dynamic control over the allocation, association, and deallocation of pointer targets is provided by the ALLOCATE, NULLIFY, and DEALLOCATE statements and pointer assignment. ALLOCATE (6.3.1) creates targets for pointers; pointer assignment (7.4.2) associates pointers with existing targets; NULLIFY (6.3.2) disassociates pointers from targets, and DEALLOCATE (6.3.3) deallocates targets. Dynamic association applies to scalars and arrays of any type.

The ALLOCATE and DEALLOCATE statements also are used to create and deallocate variables with the ALLOCATABLE attribute.

NOTE 6.16

Detailed remarks regarding pointers and dynamic association are in C.3.3.

6.3.1 ALLOCATE statement

The **ALLOCATE statement** dynamically creates pointer targets and allocatable variables.

R623	*allocate-stmt*	**is**	ALLOCATE ([*type-spec* ::] *allocation-list* ■
			■ [, *alloc-opt-list*])
R624	*alloc-opt*	**is**	STAT = *stat-variable*
		or	ERRMSG = *errmsg-variable*
		or	SOURCE = *source-expr*
R625	*stat-variable*	**is**	*scalar-int-variable*
R626	*errmsg-variable*	**is**	*scalar-default-char-variable*
R627	*source-expr*	**is**	*expr*
R628	*allocation*	**is**	*allocate-object* [(*allocate-shape-spec-list*)]

R629	*allocate-object*	**is**	*variable-name*
		or	*structure-component*
R630	*allocate-shape-spec*	**is**	[*lower-bound-expr* :] *upper-bound-expr*
R631	*lower-bound-expr*	**is**	*scalar-int-expr*
R632	*upper-bound-expr*	**is**	*scalar-int-expr*

C622 (R629) Each ***allocate-object*** shall be a nonprocedure pointer or an allocatable variable.

C623 (R623) If any ***allocate-object*** in the statement has a deferred type parameter, either ***type-spec*** or SOURCE= shall appear.

C624 (R623) If a ***type-spec*** appears, it shall specify a type with which each ***allocate-object*** is type compatible.

C625 (R623) If any ***allocate-object*** is unlimited polymorphic, either ***type-spec*** or SOURCE= shall appear.

C626 (R623) A ***type-param-value*** in a ***type-spec*** shall be an asterisk if and only if each ***allocate-object*** is a dummy argument for which the corresponding type parameter is assumed.

C627 (R623) If a ***type-spec*** appears, the kind type parameter values of each ***allocate-object*** shall be the same as the corresponding type parameter values of the ***type-spec***.

C628 (R628) An ***allocate-shape-spec-list*** shall appear if and only if the ***allocate-object*** is an array.

C629 (R628) The number of ***allocate-shape-spec***s in an ***allocate-shape-spec-list*** shall be the same as the rank of the ***allocate-object***.

C630 (R624) No ***alloc-opt*** shall appear more than once in a given ***alloc-opt-list***.

C631 (R623) If SOURCE= appears, ***type-spec*** shall not appear and ***allocation-list*** shall contain only one ***allocate-object***, which shall be type compatible (5.1.1.2) with ***source-expr***.

C632 (R623) The ***source-expr*** shall be a scalar or have the same rank as ***allocate-object***.

C633 (R623) Corresponding kind type parameters of ***allocate-object*** and ***source-expr*** shall have the same values.

An ***allocate-object*** or a bound or type parameter of an ***allocate-object*** shall not depend on the value of ***stat-variable***, the value of ***errmsg-variable***, or on the value, bounds, length type parameters, allocation status, or association status of any ***allocate-object*** in the same ALLOCATE statement.

Neither ***stat-variable***, ***source-expr***, nor ***errmsg-variable*** shall be allocated within the ALLOCATE statement in which it appears; nor shall they depend on the value, bounds, length type parameters, allocation status, or association status of any ***allocate-object*** in the same ALLOCATE statement.

The optional ***type-spec*** specifies the dynamic type and type parameters of the objects to be allocated. If a ***type-spec*** is specified, allocation of a polymorphic object allocates an object with the specified dynamic type; if a ***source-expr*** is specified, the allocation allocates an object whose dynamic type and type parameters are the same as those of the ***source-expr***; otherwise it allocates an object with a dynamic type the same as its declared type.

When an ALLOCATE statement having a ***type-spec*** is executed, any ***type-param-value***s in the ***type-spec*** specify the type parameters. If the value specified for a type parameter differs from a corresponding nondeferred value specified in the declaration of any of the ***allocate-object***s then an error condition occurs.

If a ***type-param-value*** in a ***type-spec*** in an ALLOCATE statement is an asterisk, it denotes the current value of that assumed type parameter. If it is an expression, subsequent redefinition or undefinition of any entity in the expression does not affect the type parameter value.

NOTE 6.17

An example of an ALLOCATE statement is:

```
ALLOCATE (X (N), B (-3 : M, 0:9), STAT = IERR_ALLOC)
```

When an ALLOCATE statement is executed for an array, the values of the lower bound and upper bound expressions determine the bounds of the array. Subsequent redefinition or undefinition of any entities in the bound expressions do not affect the array bounds. If the lower bound is omitted, the default value is 1. If the upper bound is less than the lower bound, the extent in that dimension is zero and the array has zero size.

NOTE 6.18

An *allocate-object* may be of type character with zero character length.

If SOURCE= appears, *source-expr* shall be conformable (2.4.5) with *allocation*. If the value of a non-deferred length type parameter of *allocate-object* is different from the value of the corresponding type parameter of *source-expr*, an error condition occurs. If the allocation is successful, the value of *allocate-object* becomes that of *source-expr*.

NOTE 6.19

An example of an ALLOCATE statement in which the value and dynamic type are determined by reference to another object is:

```
CLASS(*), ALLOCATABLE :: NEW
CLASS(*), POINTER :: OLD
! ...
ALLOCATE (NEW, SOURCE=OLD)  ! Allocate NEW with the value and dynamic type of OLD
```

A more extensive example is given in C.3.2.

If the STAT= specifier appears, successful execution of the ALLOCATE statement causes the *stat-variable* to become defined with a value of zero. If an error condition occurs during the execution of the ALLOCATE statement, the *stat-variable* becomes defined with a processor-dependent positive integer value and each *allocate-object* will have a processor-dependent status; each *allocate-object* that was successfully allocated shall have an allocation status of allocated or a pointer association status of associated; each *allocate-object* that was not successfully allocated shall retain its previous allocation status or pointer association status.

If an error condition occurs during execution of an ALLOCATE statement that does not contain the STAT= specifier, execution of the program is terminated.

The ERRMSG= specifier is described in 6.3.1.3.

6.3.1.1 Allocation of allocatable variables

The allocation status of an allocatable entity is one of the following at any time during the execution of a program:

(1) The status of an allocatable variable becomes **allocated** if it is allocated by an ALLOCATE statement, if it is allocated during assignment, or if it is given that status by the allocation transfer procedure (13.5.16). An allocatable variable with this status may be referenced, defined, or deallocated; allocating it causes an error condition in the ALLOCATE statement.

The intrinsic function ALLOCATED (13.7.9) returns true for such a variable.

(2) An allocatable variable has a status of **unallocated** if it is not allocated. The status of an allocatable variable becomes unallocated if it is deallocated (6.3.3) or if it is given that status by the allocation transfer procedure. An allocatable variable with this status shall not be referenced or defined. It shall not be supplied as an actual argument corresponding to a nonallocatable dummy argument, except to certain intrinsic inquiry functions. It may be allocated with the ALLOCATE statement. Deallocating it causes an error condition in the DEALLOCATE statement. The intrinsic function ALLOCATED (13.7.9) returns false for such a variable.

At the beginning of execution of a program, allocatable variables are unallocated.

A saved allocatable object has an initial status of unallocated. The status may change during the execution of the program.

When the allocation status of an allocatable variable changes, the allocation status of any associated allocatable variable changes accordingly. Allocation of an allocatable variable establishes values for the deferred type parameters of all associated allocatable variables.

An unsaved allocatable object that is a local variable of a procedure has a status of unallocated at the beginning of each invocation of the procedure. The status may change during execution of the procedure. An unsaved allocatable object that is a local variable of a module or a subobject thereof has an initial status of unallocated. The status may change during execution of the program.

When an object of derived type is created by an ALLOCATE statement, any allocatable ultimate components have an allocation status of unallocated.

6.3.1.2 Allocation of pointer targets

Allocation of a pointer creates an object that implicitly has the TARGET attribute. Following successful execution of an ALLOCATE statement for a pointer, the pointer is associated with the target and may be used to reference or define the target. Additional pointers may become associated with the pointer target or a part of the pointer target by pointer assignment. It is not an error to allocate a pointer that is already associated with a target. In this case, a new pointer target is created as required by the attributes of the pointer and any array bounds, type, and type parameters specified by the ALLOCATE statement. The pointer is then associated with this new target. Any previous association of the pointer with a target is broken. If the previous target had been created by allocation, it becomes inaccessible unless other pointers are associated with it. The ASSOCIATED intrinsic function (13.7.13) may be used to determine whether a pointer that does not have undefined association status is associated.

At the beginning of execution of a function whose result is a pointer, the association status of the result pointer is undefined. Before such a function returns, it shall either associate a target with this pointer or cause the association status of this pointer to become defined as disassociated.

6.3.1.3 ERRMSG= specifier

If an error condition occurs during execution of an ALLOCATE or DEALLOCATE statement, the processor shall assign an explanatory message to *errmsg-variable*. If no such condition occurs, the processor shall not change the value of *errmsg-variable*.

6.3.2 NULLIFY statement

The **NULLIFY statement** causes pointers to be disassociated.

R633	*nullify-stmt*	**is**	NULLIFY (*pointer-object-list*)
R634	*pointer-object*	**is**	*variable-name*

or *structure-component*
or *proc-pointer-name*

C634 (R634) Each *pointer-object* shall have the POINTER attribute.

A *pointer-object* shall not depend on the value, bounds, or association status of another *pointer-object* in the same NULLIFY statement.

NOTE 6.20

When a NULLIFY statement is applied to a polymorphic pointer (5.1.1.2), its dynamic type becomes the declared type.

6.3.3.1 Deallocation of allocatable variables

Deallocating an unallocated allocatable variable causes an error condition in the DEALLOCATE statement. Deallocating an allocatable variable with the TARGET attribute causes the pointer association status of any pointer associated with it to become undefined.

When the execution of a procedure is terminated by execution of a RETURN or END statement, an allocatable variable that is a named local variable of the procedure retains its allocation and definition status if it has the SAVE attribute or is a function result variable or a subobject thereof; otherwise, it is deallocated.

NOTE 6.23

The ALLOCATED intrinsic function may be used to determine whether a variable is allocated or unallocated.

If an unsaved allocatable object is a local variable of a module, and it is allocated when execution of a RETURN or END statement results in no active scoping unit having access to the module, it is processor-dependent whether the object retains its allocation status or is deallocated.

NOTE 6.24

The following example illustrates the effects of SAVE on allocation status.

```
MODULE MOD1
TYPE INITIALIZED_TYPE
   INTEGER :: I = 1  ! Default initialization
END TYPE INITIALIZED_TYPE
SAVE :: SAVED1, SAVED2
INTEGER :: SAVED1, UNSAVED1
TYPE(INITIALIZED_TYPE) :: SAVED2, UNSAVED2
ALLOCATABLE :: SAVED1(:), SAVED2(:), UNSAVED1(:), UNSAVED2(:)
END MODULE MOD1
PROGRAM MAIN
CALL SUB1   ! The values returned by the ALLOCATED intrinsic calls
            ! in the PRINT statement are:
            ! .FALSE., .FALSE., .FALSE., and .FALSE.
            ! Module MOD1 is used, and its variables are allocated.
            ! After return from the subroutine, whether the variables
            ! which were not specified with the SAVE attribute
            ! retain their allocation status is processor dependent.
CALL SUB1   ! The values returned by the first two ALLOCATED intrinsic
            ! calls in the PRINT statement are:
            ! .TRUE., .TRUE.
            ! The values returned by the second two ALLOCATED
            ! intrinsic calls in the PRINT statement are
            ! processor dependent and each could be either
            ! .TRUE. or .FALSE.
CONTAINS
   SUBROUTINE SUB1
   USE MOD1    ! Brings in saved and unsaved variables.
   PRINT *, ALLOCATED(SAVED1), ALLOCATED(SAVED2), &
            ALLOCATED(UNSAVED1), ALLOCATED(UNSAVED2)
   IF (.NOT. ALLOCATED(SAVED1)) ALLOCATE(SAVED1(10))
   IF (.NOT. ALLOCATED(SAVED2)) ALLOCATE(SAVED2(10))
   IF (.NOT. ALLOCATED(UNSAVED1)) ALLOCATE(UNSAVED1(10))
```

NOTE 6.24 (cont.)

```
   IF (.NOT. ALLOCATED(UNSAVED2)) ALLOCATE(UNSAVED2(10))
   END SUBROUTINE SUB1
END PROGRAM MAIN
```

If an executable construct references a function whose result is either allocatable or a structure with a subobject that is allocatable, and the function reference is executed, an allocatable result and any subobject that is an allocated allocatable entity in the result returned by the function is deallocated after execution of the innermost executable construct containing the reference.

If a specification expression in a scoping unit references a function whose result is either allocatable or a structure with a subobject that is allocatable, and the function reference is executed, an allocatable result and any subobject that is an allocated allocatable entity in the result returned by the function is deallocated before execution of the first executable statement in the scoping unit.

When a procedure is invoked, any allocated allocatable object that is an actual argument associated with an INTENT(OUT) allocatable dummy argument is deallocated; any allocated allocatable object that is a subobject of an actual argument associated with an INTENT(OUT) dummy argument is deallocated.

When an intrinsic assignment statement (7.4.1.3) is executed, any allocated allocatable subobject of the *variable* is deallocated before the assignment takes place.

When a variable of derived type is deallocated, any allocated allocatable subobject is deallocated.

If an allocatable component is a subobject of a finalizable object, that object is finalized before the component is automatically deallocated.

The effect of automatic deallocation is the same as that of a DEALLOCATE statement without a *dealloc-opt-list*.

NOTE 6.25

In the following example:

```
SUBROUTINE PROCESS
  REAL, ALLOCATABLE :: TEMP(:)
  REAL, ALLOCATABLE, SAVE :: X(:)
  ...
END SUBROUTINE PROCESS
```

on return from subroutine PROCESS, the allocation status of X is preserved because X has the SAVE attribute. TEMP does not have the SAVE attribute, so it will be deallocated. On the next invocation of PROCESS, TEMP will have an allocation status of unallocated.

6.3.3.2 Deallocation of pointer targets

If a pointer appears in a DEALLOCATE statement, its association status shall be defined. Deallocating a pointer that is disassociated or whose target was not created by an ALLOCATE statement causes an error condition in the DEALLOCATE statement. If a pointer is associated with an allocatable entity, the pointer shall not be deallocated.

If a pointer appears in a DEALLOCATE statement, it shall be associated with the whole of an object that was created by allocation. Deallocating a pointer target causes the pointer association status of any other pointer that is associated with the target or a portion of the target to become undefined.

Section 7: Expressions and assignment

This section describes the formation, interpretation, and evaluation rules for expressions, intrinsic and defined assignment, pointer assignment, masked array assignment (WHERE), and FORALL.

7.1 Expressions

An **expression** represents either a data reference or a computation, and its value is either a scalar or an array. An expression is formed from operands, operators, and parentheses.

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

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

7.1.1 Form of an expression

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

These categories are related to the different operator precedence levels and, in general, are 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.

7.1.1.1 Primary

R701	*primary*	**is**	*constant*
		or	*designator*
		or	*array-constructor*
		or	*structure-constructor*
		or	*function-reference*
		or	*type-param-inquiry*
		or	*type-param-name*
		or	(*expr*)

C701 (R701) The *type-param-name* shall be the name of a type parameter.

C702 (R701) The *designator* shall not be a whole assumed-size array.

NOTE 7.1

Examples of a *primary* are:

Example	Syntactic class
`1.0`	*constant*
`'ABCDEFGHIJKLMNOPQRSTUVWXYZ' (I:I)`	*constant-subobject*
`A`	*variable*
`(/ 1.0, 2.0 /)`	*array-constructor*
`PERSON (12, 'Jones')`	*structure-constructor*
`F (X, Y)`	*function-reference*

NOTE 7.1 (cont.)

`(S + T)`	(*expr*)

7.1.1.2 Level-1 expressions

Defined unary operators have the highest operator precedence (Table 7.7). Level-1 expressions are primaries optionally operated on by defined unary operators:

R702 *level-1-expr* **is** [*defined-unary-op*] *primary*
R703 *defined-unary-op* **is** . *letter* [*letter*] ... .

C703 (R703) A *defined-unary-op* shall not contain more than 63 letters and shall not be the same as any *intrinsic-operator* or *logical-literal-constant*.

NOTE 7.2

Simple examples of a level-1 expression are:

Example	Syntactic class
`A`	*primary* (R701)
`.INVERSE. B`	*level-1-expr* (R702)

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

```
.INVERSE. (A
```

NOTE 7.3 (cont.)

```
- A + D * E + B ** C
```

7.1.1.4 Level-3 expressions

Level-3 expressions are level-2 expressions optionally involving the character operator *concat-op*.

R710 *level-3-expr* **is** [*level-3-expr concat-op*] *level-2-expr*
R711 *concat-op* **is** //

NOTE 7.4

Simple examples of a level-3 expression are:

Example	Syntactic class
`A`	*level-2-expr* (R706)
`B // C`	*level-3-expr* (R710)

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

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

7.1.1.5 Level-4 expressions

Level-4 expressions are level-3 expressions optionally involving the relational operators *rel-op*.

R712 *level-4-expr* **is** [*level-3-expr rel-op*] *level-3-expr*
R713 *rel-op* **is** .EQ.
or .NE.
or .LT.
or .LE.
or .GT.
or .GE.
or ==
or /=
or <
or <=
or >
or >=

NOTE 7.5

Simple examples of a level-4 expression are:

Example	Syntactic class
`A`	*level-3-expr* (R710)
`B == C`	*level-4-expr* (R712)
`D < E`	*level-4-expr* (R712)

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

```
(A + B) /= C
```

7.1.1.6 Level-5 expressions

Level-5 expressions are level-4 expressions optionally involving the logical operators *not-op*, *and-op*, *or-op*, and *equiv-op*.

R714	*and-operand*	**is**	[*not-op*] *level-4-expr*
R715	*or-operand*	**is**	[*or-operand and-op*] *and-operand*
R716	*equiv-operand*	**is**	[*equiv-operand or-op*] *or-operand*
R717	*level-5-expr*	**is**	[*level-5-expr equiv-op*] *equiv-operand*
R718	*not-op*	**is**	.NOT.
R719	*and-op*	**is**	.AND.
R720	*or-op*	**is**	.OR.
R721	*equiv-op*	**is**	.EQV.
		or	.NEQV.

NOTE 7.6

Simple examples of a level-5 expression are:

Example	Syntactic class
`A`	*level-4-expr* (R712)
`.NOT. B`	*and-operand* (R714)
`C .AND. D`	*or-operand* (R715)
`E .OR. F`	*equiv-operand* (R716)
`G .EQV. H`	*level-5-expr* (R717)
`S .NEQV. T`	*level-5-expr* (R717)

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

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

7.1.1.7 General form of an expression

Expressions are level-5 expressions optionally involving defined binary operators. Defined binary operators have the lowest operator precedence (Table 7.7).

R722	*expr*	**is**	[*expr defined-binary-op*] *level-5-expr*
R723	*defined-binary-op*	**is**	. *letter* [*letter*] ... .

C704 (R723) A *defined-binary-op* shall not contain more than 63 letters and shall not be the same as any *intrinsic-operator* or *logical-literal-constant*.

NOTE 7.7

Simple examples of an expression are:

Example	Syntactic class
`A`	*level-5-expr* (R717)
`B.UNION.C`	*expr* (R722)

More complicated examples of an expression are:

```
(B .INTERSECT. C) .UNION. (X - Y)
A + B == C * D
.INVERSE. (A + B)
A + B .AND. C * D
E // G == 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 (4.4) listed in Table 7.1 for the unary intrinsic operator.

An **intrinsic binary operation** is an operation of the form x_1 *intrinsic-operator* x_2 where x_1 and x_2 are of the intrinsic types (4.4) listed in Table 7.1 for the binary intrinsic operator and are in shape conformance (7.1.5).

Table 7.1: **Type of operands and results for intrinsic operators**

Intrinsic operator *op*	Type of x_1	Type of x_2	Type of $[x_1]$ *op* x_2
Unary +, −		I, R, Z	I, R, Z
Binary +, −, *, /, **	I	I, R, Z	I, R, Z
	R	I, R, Z	R, R, Z
	Z	I, R, Z	Z, Z, Z
//	C	C	C
.EQ., .NE., ==, /=	I	I, R, Z	L, L, L
	R	I, R, Z	L, L, L
	Z	I, R, Z	L, L, L
	C	C	L
.GT., .GE., .LT., .LE. >, >=, <, <=	I	I, R	L, L
	R	I, R	L, L
	C	C	L
.NOT.		L	L
.AND., .OR., .EQV., .NEQV.	L	L	L

Note: The symbols I, R, Z, C, and L stand for the types integer, real, complex, character, and logical, 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. For the intrinsic operators with operands of type character, the kind type parameters of the operands shall be the same.

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 kind type parameters. Except for a value raised to an integer power, if the operands have different types or kind type parameters, the effect is as if each operand that differs in type or kind type parameter from those of the result is converted to the type and kind type parameter of the result before the operation is performed. When a value of type real or complex is raised to an integer power, the integer operand need not be converted.

A **character intrinsic operation**, **relational intrinsic operation**, and **logical intrinsic operation** are similarly defined in terms of a **character intrinsic operator** (//), **relational intrinsic operator** (.EQ., .NE., .GT., .GE., .LT., .LE., ==, /=, >, >=, <, and <=), and **logical intrinsic operator** (.AND., .OR., .NOT., .EQV., and .NEQV.), respectively. For the character intrinsic operator //, the kind type parameters shall be the same. For the relational intrinsic operators with character operands, the kind type parameters shall be the same.

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.

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 that has the form *defined-unary-op* x_2 or *intrinsic-operator* x_2 and that is defined by a function and a generic interface (4.5.1, 12.3.2.1).

A function defines the unary operation *op* x_2 if

(1) The function is specified with a FUNCTION (12.5.2.1) or ENTRY (12.5.2.4) statement that specifies one dummy argument d_2,

(2) Either

  (a) A generic interface (12.3.2.1) provides the function with a *generic-spec* of OPERATOR (*op*), or

  (b) There is a generic binding (4.5.1) in the declared type of x_2 with a *generic-spec* of OPERATOR (*op*) and there is a corresponding binding to the function in the dynamic type of x_2,

(3) The type of d_2 is compatible with the dynamic type of x_2,

(4) The type parameters, if any, of d_2 match the corresponding type parameters of x_2, and

(5) Either

  (a) The rank of x_2 matches that of d_2 or

  (b) The function is elemental and there is no other function that defines the operation.

If d_2 is an array, the shape of x_2 shall match the shape of d_2.

A **defined binary operation** is an operation that has the form x_1 *defined-binary-op* x_2 or x_1 *intrinsic-operator* x_2 and that is defined by a function and a generic interface.

A function defines the binary operation x_1 *op* x_2 if

(1) The function is specified with a FUNCTION (12.5.2.1) or ENTRY (12.5.2.4) statement that specifies two dummy arguments, d_1 and d_2,

(2) Either

  (a) A generic interface (12.3.2.1) provides the function with a *generic-spec* of OPERATOR (*op*), or

  (b) There is a generic binding (4.5.1) in the declared type of x_1 or x_2 with a *generic-spec* of OPERATOR (*op*) and there is a corresponding binding to the function in the dynamic type of x_1 or x_2, respectively,

(3) The types of d_1 and d_2 are compatible with the dynamic types of x_1 and x_2, respectively,

(4) The type parameters, if any, of d_1 and d_2 match the corresponding type parameters of x_1 and x_2, respectively, and

(5) Either

  (a) The ranks of x_1 and x_2 match those of d_1 and d_2 or

  (b) The function is elemental, x_1 and x_2 are conformable, and there is no other function that defines the operation.

If d_1 or d_2 is an array, the shapes of x_1 and x_2 shall match the shapes of d_1 and d_2, respectively.

NOTE 7.8

An intrinsic operator may be used as the operator in a defined operation. In such a case, the generic properties of the operator are extended.

An **extension operation** is a defined operation in which the operator is of the form *defined-unary-op* or *defined-binary-op*. Such an operator is called an **extension operator**. The operator used in an extension operation may be such that a generic interface for the operator may specify more than one function.

A **defined elemental operation** is a defined operation for which the function is elemental (12.7).

7.1.4 Type, type parameters, and shape of an expression

The type, type parameters, and shape of an expression depend on the operators and on the types, type parameters, and shapes of the primaries used in the expression, and are determined recursively from the syntactic form of the expression. The type of an expression is one of the intrinsic types (4.4) or a derived type (4.5).

If an expression is a polymorphic primary or defined operation, the type parameters and the declared and dynamic types of the expression are the same as those of the primary or defined operation. Otherwise the type parameters and dynamic type of the expression are the same as its declared type and type parameters; they are referred to simply as the type and type parameters of the expression.

R724 *logical-expr* **is** *expr*

C705 (R724) *logical-expr* shall be of type logical.

R725 *char-expr* **is** *expr*

C706 (R725) *char-expr* shall be of type character.

R726 *default-char-expr* **is** *expr*

C707 (R726) *default-char-expr* shall be of type default character.

R727 *int-expr* **is** *expr*

C708 (R727) *int-expr* shall be of type integer.

R728 *numeric-expr* **is** *expr*

C709 (R728) *numeric-expr* shall be of type integer, real, or complex.

7.1.4.1 Type, type parameters, and shape of a primary

The type, type parameters, and shape of a primary are determined according to whether the primary is a constant, variable, array constructor, structure constructor, function reference, type parameter inquiry, type parameter name, or parenthesized expression. If a primary is a constant, its type, type parameters, and shape are those of the constant. If it is a structure constructor, it is scalar and its type and type parameters are as described in 4.5.9. If it is an array constructor, its type, type parameters, and shape are as described in 4.7. If it is a variable or function reference, its type, type parameters, and shape are those of the variable (5.1.1, 5.1.2) or the function reference (12.4.2), respectively. If the function reference is generic (12.3.2.1, 13.5) then its type, type parameters, and shape are those of the specific function referenced, which is determined by the types, type parameters, and ranks of its actual arguments as specified in 16.2.3. If it is a type parameter inquiry or type parameter name, it is a scalar integer with the kind of the type parameter.

If a primary is a parenthesized expression, its type, type parameters, and shape are those of the expression.

If a pointer appears as one of the following, the associated target object is referenced:

(1) A primary in an intrinsic or defined operation,
(2) The *expr* of a parenthesized primary, or
(3) The only primary on the right-hand side of an intrinsic assignment statement.

The type, type parameters, and shape of the primary are those of the current target. If the pointer is not associated with a target, it may appear as a primary only as an actual argument in a reference to a procedure whose corresponding dummy argument is declared to be a pointer, or as the target in a pointer assignment statement.

A disassociated array pointer or an unallocated allocatable array has no shape but does have rank. The type, type parameters, and rank of the result of the NULL intrinsic function depend on context (13.7.88).

7.1.4.2 Type, type parameters, and shape of the result of an operation

The type of the result of an intrinsic operation $[x_1]$ *op* x_2 is specified by Table 7.1. The shape of the result of an intrinsic operation is the shape of x_2 if *op* is unary or if x_1 is scalar, and is the shape of x_1 otherwise.

The type, type parameters, and shape of the result of a defined operation $[x_1]$ *op* x_2 is specified by the function defining the operation (7.2).

An expression of an intrinsic type has a kind type parameter. An expression of type character also has a character length parameter.

The type parameters of the result of an intrinsic operation are as follows:

(1) For an expression x_1 // x_2 where // is the character intrinsic operator and x_1 and x_2 are of type character, the character length parameter is the sum of the lengths of the operands and the kind type parameter is the kind type parameter of x_1, which shall be the same as the kind type parameter of x_2.
(2) For an expression *op* x_2 where *op* is an intrinsic unary operator and x_2 is of type integer, real, complex, or logical, the kind type parameter of the expression is that of the operand.
(3) 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 or complex, the kind type parameter of the expression is that of the real or complex operand.
(4) For an expression x_1 *op* x_2 where *op* is a numeric intrinsic binary operator with both operands of the same type and kind type parameters, or with one real and one complex with the same kind type parameters, the kind type parameter of the expression is identical to that of each operand. In the case where both operands are integer with different kind type parameters, the kind type parameter of the expression is that of the operand with the greater decimal exponent range if the decimal exponent ranges are different; if the decimal exponent ranges are the same, the kind type parameter of the expression is processor dependent, but it is the same as that of one of the operands. In the case where both operands are any of type real or complex with different kind type parameters, the kind type parameter of the expression is that of the operand with the greater decimal precision if the decimal precisions are different; if the decimal precisions are the same, the kind type parameter of the expression is processor dependent, but it is the same as that of one of the operands.
(5) For an expression x_1 *op* x_2 where *op* is a logical intrinsic binary operator with both operands of the same kind type parameter, the kind type parameter of the expression is identical to that of each operand. In the case where both operands are of type logical with different kind type parameters, the kind type parameter of the expression is processor dependent, but it is the same as that of one of the operands.
(6) For an expression x_1 *op* x_2 where *op* is a relational intrinsic operator, the expression has the default logical kind type parameter.

7.1.5 Conformability rules for elemental operations

An **elemental operation** is an intrinsic operation or a defined elemental operation. Two entities are in **shape conformance** if both are arrays of the same shape, or one or both are scalars.

For all elemental binary operations, the two operands shall be in shape conformance. In the case where 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, if any, of the array equal to the value of the scalar.

7.1.6 Specification expression

A **specification expression** is an expression with limitations that make it suitable for use in specifications such as length type parameters (C501) and array bounds (R512, R513).

R729 *specification-expr* **is** *scalar-int-expr*

C710 (R729) The *scalar-int-expr* shall be a restricted expression.

A **restricted expression** is an expression in which each operation is intrinsic and each primary is

(1) A constant or subobject of a constant,
(2) An object designator with a base object that is a dummy argument that has neither the OPTIONAL nor the INTENT (OUT) attribute,
(3) An object designator with a base object that is in a common block,
(4) An object designator with a base object that is made accessible by use association or host association,
(5) An array constructor where each element and each *scalar-int-expr* of each *ac-implied-do-control* is a restricted expression,
(6) A structure constructor where each component is a restricted expression,
(7) A specification inquiry where each designator or function argument is
    (a) a restricted expression or
    (b) a variable whose properties inquired about are not
        (i) dependent on the upper bound of the last dimension of an assumed-size array,
        (ii) deferred, or
        (iii) defined by an expression that is not a restricted expression,
(8) A reference to any other standard intrinsic function where each argument is a restricted expression,
(9) A reference to a specification function where each argument is a restricted expression,
(10) A type parameter of the derived type being defined,
(11) An *ac-do-variable* within an array constructor where each *scalar-int-expr* of the corresponding *ac-implied-do-control* is a restricted expression, or
(12) A restricted expression enclosed in parentheses,

where each subscript, section subscript, substring starting point, substring ending point, and type parameter value is a restricted expression, and where any final subroutine that is invoked is pure.

A **specification inquiry** is a reference to

(1) an array inquiry function (13.5.7),
(2) the bit inquiry function BIT_SIZE,
(3) the character inquiry function LEN,
(4) the kind inquiry function KIND,

(5) the character inquiry function NEW_LINE,
(6) a numeric inquiry function (13.5.6),
(7) a type parameter inquiry (6.1.3), or
(8) an IEEE inquiry function (14.9.1),

A function is a **specification function** if it is a pure function, is not a standard intrinsic function, is not an internal function, is not a statement function, and does not have a dummy procedure argument.

Evaluation of a specification expression shall not directly or indirectly cause a procedure defined by the subprogram in which it appears to be invoked.

NOTE 7.9

Specification functions are nonintrinsic functions that may be used in specification expressions to determine the attributes of data objects. The requirement that they be pure ensures that they cannot have side effects that could affect other objects being declared in the same *specification-part*. The requirement that they not be internal ensures that they cannot inquire, via host association, about other objects being declared in the same *specification-part*. The prohibition against recursion avoids the creation of a new instance of a procedure while construction of one is in progress.

A variable in a specification expression shall have its type and type parameters, if any, specified by a previous declaration in the same scoping unit, by the implicit typing rules in effect for the scoping unit, or by host or use association. If a variable in a specification expression is typed by the implicit typing rules, its appearance in any subsequent type declaration statement shall confirm the implied type and type parameters.

If a specification expression includes a specification inquiry that depends on a type parameter or an array bound of an entity specified in the same *specification-part*, the type parameter or array bound shall be specified in a prior specification of the *specification-part*. The prior specification may be to the left of the specification inquiry in the same statement, but shall not be within the same *entity-decl*. If a specification expression includes a reference to the value of an element of an array specified in the same *specification-part*, the array shall be completely specified in prior declarations.

NOTE 7.10

The following are examples of specification expressions:

```
LBOUND (B, 1) + 5   ! B is an assumed-shape dummy array
M + LEN (C)         ! M and C are dummy arguments
2 * PRECISION (A)   ! A is a real variable made accessible
                    ! by a USE statement
```

7.1.7 Initialization expression

An **initialization expression** is an expression with limitations that make it suitable for use as a kind type parameter, initializer, or named constant. It is an expression in which each operation is intrinsic, and each primary is

(1) A constant or subobject of a constant,
(2) An array constructor where each element and each *scalar-int-expr* of each *ac-implied-do-control* is an initialization expression,
(3) A structure constructor where each *component-spec* corresponding to an allocatable com-

ization expression,

(5) A reference to a transformational standard intrinsic function other than NULL, where each argument is an initialization expression,

(6) A reference to the transformational intrinsic function NULL that does not have an argument with a type parameter that is assumed or is defined by an expression that is not an initialization expression,

(7) A reference to the transformational function IEEE_SELECTED_REAL_KIND from the intrinsic module IEEE_ARITHMETIC (14), where each argument is an initialization expression.

(8) A specification inquiry where each designator or function argument is
   (a) an initialization expression or
   (b) a variable whose properties inquired about are not
      (i) assumed,
      (ii) deferred, or
      (iii) defined by an expression that is not an initialization expression,

(9) A kind type parameter of the derived type being defined,

(10) An *ac-do-variable* within an array constructor where each *scalar-int-expr* of the corresponding *ac-implied-do-control* is an initialization expression, or

(11) An initialization expression enclosed in parentheses,

and where each subscript, section subscript, substring starting point, substring ending point, and type parameter value is an initialization expression.

R730 *initialization-expr* **is** *expr*

C711 (R730) *initialization-expr* shall be an initialization expression.

R731 *char-initialization-expr* **is** *char-expr*

C712 (R731) *char-initialization-expr* shall be an initialization expression.

R732 *int-initialization-expr* **is** *int-expr*

C713 (R732) *int-initialization-expr* shall be an initialization expression.

R733 *logical-initialization-expr* **is** *logical-expr*

C714 (R733) *logical-initialization-expr* shall be an initialization expression.

If an initialization expression includes a specification inquiry that depends on a type parameter or an array bound of an entity specified in the same *specification-part*, the type parameter or array bound shall be specified in a prior specification of the *specification-part*. The prior specification may be to the left of the specification inquiry in the same statement, but shall not be within the same *entity-decl*.

NOTE 7.11

The following are examples of initialization expressions:

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

NOTE 7.11 (cont.)

```
4.0 * atan(1.0)
ceiling(number_of_decimal_digits / log10(radix(0.0)))
```

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

7.1.8 Evaluation of operations

An intrinsic operation requires the values of its operands.

The execution of any numeric operation whose result is not defined by the arithmetic used by the processor is prohibited. Raising a negative-valued primary of type real to a real power is prohibited.

The evaluation of a function reference shall neither affect nor be affected by the evaluation of any other entity within the statement. If a function reference causes definition or undefinition of an actual argument of the function, that argument or any associated entities shall not appear elsewhere in the same statement. However, execution of a function reference in the logical expression in an IF statement (8.1.2.4), the mask expression in a WHERE statement (7.4.3.1), or the subscripts and strides in a FORALL statement (7.4.4) is permitted to define variables in the statement that is conditionally executed.

NOTE 7.12

For example, the statements

```
A (I) = F (I)
Y = G (X) + X
```

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

However, in the statements

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

F or G may define X.

The declared 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.

Execution of an array element reference requires the evaluation of its subscripts. The type of an expression in which the array element reference appears does not affect, and is not affected by, the evaluation of its subscripts. Execution of an array section reference requires the evaluation of its section subscripts. 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. Execution of a substring reference requires the evaluation of its substring expressions. The type of an expression in which a substring appears does not affect, and is not affected by, the evaluation of the substring expressions. It is not necessary for a processor to evaluate any subscript expressions or substring expressions for an array of zero size or a character entity of zero length.

The appearance of an array constructor requires the evaluation of each *scalar-int-expr* of the *ac-implied-do-control* in any *ac-implied-do* it may contain. The type of an expression in which an array constructor appears does not affect, and is not affected by, the evaluation of such bounds and stride expressions.

When an elemental binary operation is applied to a scalar and an array or to two arrays of the same shape, the operation is performed element-by-element on corresponding array elements of the array operands. The processor may perform the element-by-element operations in any order.

NOTE 7.13

For example, the array expression

```
A + B
```

produces an array of 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.

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

7.1.8.1 Evaluation of operands

It is not necessary for a processor to evaluate all of the operands of an expression, or to evaluate entirely each operand, if the value of the expression can be determined otherwise.

NOTE 7.14

This principle is most often applicable to logical expressions, zero-sized arrays, and zero-length strings, but it applies to all expressions.

For example, in evaluating the expression

```
X > 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

```
W (Z) + A
```

where A 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.

NOTE 7.15

In the examples in Note 7.14, if `L` or `W` defines its argument, evaluation of the expressions under the specified conditions causes `Z` to become undefined, no matter whether or not `L`(Z) or `W`(Z) is evaluated.

7.1.8.2 Integrity of parentheses

The sections that follow state certain conditions under which a processor may evaluate an expression that is different from the one specified by applying the rules given in 7.1.1 and 7.2. However, any expression in parentheses shall be treated as a data entity.

NOTE 7.16

For example, in evaluating the expression A + (B – C) where A, B, and C are of numeric types, the difference of B and C shall be evaluated before the addition operation is performed; the processor

NOTE 7.16 (cont.)

shall not evaluate the mathematically equivalent expression (A + B) – C.

7.1.8.3 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 numeric type may produce different computational results.

NOTE 7.17

Any difference between the values of the expressions (1./3.)*3. and 1. is a computational difference, not a mathematical difference. The difference between the values of the expressions 5/2 and 5./2. is a mathematical difference, not a computational difference.

The mathematical definition of integer division is given in 7.2.1.1.

NOTE 7.18

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 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 shall not be used by a processor in the evaluation of those expressions.

Expression	Forbidden alternative form
I / 2	0.5 * I
X * I / J	X * (I / J)
I / J / A	I / (J * A)
(X + Y) + Z	X + (Y + Z)
(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.

NOTE 7.19

For example, in the expression

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

the parenthesized expression (B – C) shall be evaluated and then added to A.

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 type that may depend on the order of evaluation used by the processor.

NOTE 7.20

For example, in the evaluation of the expression

```
Z + R + I
```

where Z, R, and I represent data objects of complex, real, and integer type, respectively, the 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.

7.1.8.4 Evaluation of the character intrinsic operation

The rules given in 7.2.2 specify the interpretation of the character intrinsic operation. A processor is only required to evaluate as much of the character intrinsic operation as is required by the context in which the expression appears.

NOTE 7.21

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 because C1 and C2 both have a length of 2.

7.1.8.5 Evaluation of relational intrinsic operations

The rules given in 7.2.3 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, provided that the integrity of parentheses in any expression is not violated.

NOTE 7.22

For example, the processor may choose to evaluate the expression

NOTE 7.22 (cont.)

```
I > J
```

where I and J are integer variables, as

```
J - I < 0
```

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

7.1.8.6 Evaluation of logical intrinsic operations

The rules given in 7.2.4 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 in any expression is not violated.

NOTE 7.23

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.

7.1.8.7 Evaluation of a defined operation

The rules given in 7.2 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 operations

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

The interpretation of a defined operation is provided by the function that defines the operation. The type, type parameters and interpretation of an expression that consists of an intrinsic or defined operation are independent of the type and type parameters of the context or any larger expression in which it appears.

NOTE 7.24

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.

The operators <, <=, >, >=, ==, and /= always have the same interpretations as the operators .LT., .LE., .GT., .GE., .EQ., and .NE., respectively.

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 for operands of the numeric intrinsic operations are specified in 7.1.2.

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 operation depends on the types of the operands (7.2.1.1).

If x_1 and x_2 are of type integer and x_2 has a negative value, the interpretation of x_1 ** x_2 is the same as the interpretation of 1/(x_1 ** ABS (x_2)), which is subject to the rules of integer division (7.2.1.1).

NOTE 7.25

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.

NOTE 7.26

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 x_1 ** x_2 is the principal value of $x_1^{x_2}$.

7.2.2 Character intrinsic operation

The character intrinsic operator // is used to concatenate two operands of type character with the same kind type parameter. Evaluation of the character intrinsic operation produces a result of type character.

The interpretation of the character intrinsic operator // when used to form an expression is 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 character intrinsic operator //**

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

The result of the 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.

NOTE 7.27

For example, the value of (’AB’ // ’CDE’) // ’F’ is the string ’ABCDEF’. Also, the value of ’AB’ // (’CDE’ // ’F’) is the string ’ABCDEF’.

7.2.3 Relational intrinsic operations

A relational intrinsic operation is used to compare values of two operands using the relational intrinsic operators .LT., .LE., .GT., .GE., .EQ., .NE., <, <=, >, >=, ==, and /=. The permitted types for operands of the relational intrinsic operators are specified in 7.1.2.

NOTE 7.28

As shown in Table 7.1, a relational intrinsic operator cannot be used to compare the value of an expression of a numeric type with one of type character or logical. Also, two operands of type logical cannot be compared, a complex operand may be compared with another numeric operand only when the operator is .EQ., .NE., ==, or /=, and two character operands cannot be compared unless they have the same kind type parameter value.

Evaluation of a relational intrinsic operation produces a result of type default logical.

The interpretation of the relational intrinsic operators is given in Table 7.4, 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 7.4: **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

Interpretation of the relational intrinsic operators (cont.)

Operator	Representing	Use of operator	Interpretation
/=	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.

In the numeric relational operation

x_1 *rel-op* x_2

if the types or kind type parameters of x_1 and x_2 differ, their values are converted to the type and kind type parameter of the expression $x_1 + x_2$ before evaluation.

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. 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. If both x_1 and x_2 are of zero length, x_1 is equal to x_2; if every character of x_1 is the same as the character in the corresponding position in x_2, x_1 is equal to x_2. Otherwise, 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 (4.4.4.3); 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.

NOTE 7.29

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.

For nondefault character types, the blank padding character is processor dependent.

7.2.4 Logical intrinsic operations

A logical operation is used to express a logical computation. Evaluation of a logical operation produces a result of type logical. The permitted types 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 7.5, 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 logical intrinsic operators**

Operator	Representing	Use of operator	Interpretation
.NOT.	Logical negation	.NOT. x_2	True if x_2 is false
.AND.	Logical conjunction	x_1 .AND. x_2	True if x_1 and x_2 are both true
.OR.	Logical inclusive disjunction	x_1 .OR. x_2	True if x_1 and/or x_2 is true
.NEQV.	Logical nonequivalence	x_1 .NEQV. x_2	True if either x_1 or x_2 is true, but not both

Interpretation of the logical intrinsic operators (cont.)

Operator	Representing	Use of operator	Interpretation
.EQV.	Logical equivalence	x_1 .EQV. x_2	True if both x_1 and x_2 are true or both are false

1 The values of the logical intrinsic operations are shown in Table 7.6.

Table 7.6: **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

2 7.3 Precedence of operators

3 There is a precedence among the intrinsic and extension operations corresponding to the form of expres-
4 sions specified in 7.1.1, which determines the order in which the operands are combined unless the order
5 is changed by the use of parentheses. This precedence order is summarized in Table 7.7.

Table 7.7: **Categories of operations and relative precedence**

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

6 The precedence of a defined operation is that of its operator.

NOTE 7.30

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 in determining the interpretation of the expression unless the order is changed by the use of parentheses.

NOTE 7.31

In interpreting a *level-2-expr* containing two or more binary operators + or –, each operand (*add-operand*) is combined from left to right. Similarly, the same left-to-right interpretation for a *mult-operand* in *add-operand*, as well as for other kinds of expressions, is a consequence of the general form. However, for interpreting a *mult-operand* expression when two or more exponentiation 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'
```

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. 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. The rules do allow a binary operator or an intrinsic unary operator to be followed by a defined unary operator, such as:

```
A * .INVERSE. B
- .INVERSE. (B)
```

As another example, in the expression

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

the general form implies a higher precedence for the .AND. operator than for the .OR. operator; therefore, the interpretation of the above expression is the same as the interpretation of the expression

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

NOTE 7.32

An expression may contain more than one category of operator. The logical expression

```
L .OR. A + B >= 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

NOTE 7.32 (cont.)

```
L .OR. ((A + B) >= C)
```

NOTE 7.33

If

(1) The operator ** is extended to type logical,
(2) The operator .STARSTAR. is defined to duplicate the function of ** on type real,
(3) .MINUS. is defined to duplicate the unary operator −, and
(4) L1 and L2 are type logical and X and Y are type real,

then in precedence: L1 ** L2 is higher than X * Y; X * Y is higher than X .STARSTAR. Y; and .MINUS. X is higher than –X.

7.4 Assignment

Execution of an assignment statement causes a variable to become defined or redefined. Execution of a pointer assignment statement causes a pointer to become associated with a target or causes its pointer association status to become disassociated or undefined. Execution of a WHERE statement or WHERE construct masks the evaluation of expressions and assignment of values in array assignment statements according to the value of a logical array expression. Execution of a FORALL statement or FORALL construct controls the assignment to elements of arrays by using a set of index variables and a mask expression.

7.4.1 Assignment statement

A variable may be defined or redefined by execution of an assignment statement.

7.4.1.1 General form

R734 *assignment-stmt* **is** *variable* = *expr*

C715 (R734) The *variable* in an *assignment-stmt* shall not be a whole assumed-size array.

NOTE 7.34

Examples of an assignment statement are:

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

An *assignment-stmt* shall meet the requirements of either a defined assignment statement or an intrinsic assignment statement.

7.4.1.2 Intrinsic assignment statement

An **intrinsic assignment statement** is an assignment statement that is not a defined assignment statement (7.4.1.4). In an intrinsic assignment statement, *variable* shall not be polymorphic, and

(1) If *expr* is an array then *variable* shall also be an array,
(2) Either *variable* shall be an allocatable array of the same rank as *expr* or the shapes of *variable* and *expr* shall conform, and

(3) The declared types of *variable* and *expr* shall conform as specified in Table 7.8.

Table 7.8: **Type conformance for the intrinsic assignment statement**

Type of *variable*	Type of *expr*
integer	integer, real, complex
real	integer, real, complex
complex	integer, real, complex
ISO 10646, ASCII, or default character	ISO 10646, ASCII, or default character
other character	character of the same kind type parameter as *variable*
logical	logical
derived type	same derived type and kind type parameters as *variable*; each length type parameter value shall be the same unless *variable* is allocatable and its corresponding type parameter is deferred

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. A **logical intrinsic assignment statement** is an intrinsic assignment statement for which *variable* and *expr* are of type logical. A **derived-type intrinsic assignment statement** is an intrinsic assignment statement for which *variable* and *expr* are of derived type.

An **array intrinsic assignment statement** is an intrinsic assignment statement for which *variable* is an array. The *variable* shall not be a many-one array section (6.2.2.3.2).

If *variable* is a pointer, it shall be associated with a definable target such that the type, type parameters, and shape of the target and *expr* conform.

7.4.1.3 Interpretation of intrinsic assignments

Execution of an intrinsic assignment causes, in effect, the evaluation of the expression *expr* and all expressions within *variable* (7.1.8), the possible conversion of *expr* to the type and type parameters of *variable* (Table 7.9), and the definition of *variable* with the resulting value. The execution of the assignment shall have the same effect as if the evaluation of all operations in *expr* and *variable* occurred before any portion of *variable* is defined by the assignment. The evaluation of expressions within *variable* shall neither affect nor be affected by the evaluation of *expr*. No value is assigned to *variable* if *variable* is of type character and zero length, or is an array of size zero.

If *variable* is a pointer, the value of *expr* is assigned to the target of *variable*.

If *variable* is an allocated allocatable variable, it is deallocated if *expr* is an array of different shape or any of the corresponding length type parameter values of *variable* and *expr* differ. If *variable* is or becomes an unallocated allocatable variable, then it is allocated with each deferred type parameter equal to the corresponding type parameters of *expr*, with the shape of *expr*, and with each lower bound equal to the corresponding element of LBOUND(*expr*).

NOTE 7.35

For example, given the declaration

```
CHARACTER(:),ALLOCATABLE :: NAME
```

then after the assignment statement

```
NAME = 'Dr. '//FIRST_NAME//' '//SURNAME
```

NOTE 7.35 (cont.)

NAME will have the length LEN(FIRST_NAME)+LEN(SURNAME)+5, even if it had previously been unallocated, or allocated with a different length. However, for the assignment statement

```
NAME(:) = 'Dr. '//FIRST_NAME//' '//SURNAME
```

NAME must already be allocated at the time of the assignment; the assigned value is truncated or blank padded to the previously allocated length of NAME.

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

NOTE 7.36

For example, in the character intrinsic assignment statement:

```
STRING (2:5) = STRING (1:4)
```

the assignment of the first character of STRING to the second character does not affect the evaluation of STRING (1:4). If the value of STRING prior to the assignment was 'ABCDEF', the value following the assignment is 'AABCDF'.

If *expr* 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*.

If *variable* is an array, the assignment is performed element-by-element on corresponding array elements of *variable* and *expr*.

NOTE 7.37

For example, if 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.

If C is an allocatable array of rank 1, then

```
C = PACK(ARRAY,ARRAY>0)
```

will cause C to contain all the positive elements of ARRAY in array element order; if C is not allocated or is allocated with the wrong size, it will be re-allocated to be of the correct size to hold the result of PACK.

The processor may perform the element-by-element assignment in any order.

NOTE 7.38

For example, the following program segment results in the values of the elements of array X being reversed:

```
REAL X (10)
   ...
X (1:10) = X (10:1:-1)
```

For a numeric intrinsic assignment statement, *variable* and *expr* may have different numeric types or different kind type parameters, in which case the value of *expr* is converted to the type and kind type parameter of *variable* according to the rules of Table 7.9.

Table 7.9: **Numeric conversion and the assignment statement**

Type of *variable*	Value Assigned
integer	INT (*expr*, KIND = KIND (*variable*))
real	REAL (*expr*, KIND = KIND (*variable*))
complex	CMPLX (*expr*, KIND = KIND (*variable*))
Note: The functions INT, REAL, CMPLX, and KIND are the generic functions defined in 13.7.	

For a logical intrinsic assignment statement, *variable* and *expr* may have different kind type parameters, in which case the value of *expr* is converted to the kind type parameter of *variable*.

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

(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 on the right with blanks until it is the same length as *variable*.

If *variable* and *expr* have different kind type parameters, each character *c* in *expr* is converted to the kind type parameter of *variable* by ACHAR(IACHAR(*c*),KIND(*variable*)).

NOTE 7.39

For nondefault character types, the blank padding character is processor dependent. When assigning a character expression to a variable of a different kind, each character of the expression that is not representable in the kind of the variable is replaced by a processor-dependent character.

A derived-type intrinsic assignment is performed as if each component of *variable* were assigned from the corresponding component of *expr* using pointer assignment (7.4.2) for each pointer component, defined assignment for each nonpointer nonallocatable component of a type that has a type-bound defined assignment consistent with the component, and intrinsic assignment for each other nonpointer nonallocatable component. For an allocatable component the following sequence of operations is applied:

(1) If the component of *variable* is allocated, it is deallocated.

(2) If the component of *expr* is allocated, the corresponding component of *variable* is allocated with the same dynamic type and type parameters as the component of *expr*. If it is an array, it is allocated with the same bounds. The value of the component of *expr* is then assigned to the corresponding component of *variable* using defined assignment if the declared type of the component has a type-bound defined assignment consistent with the component, and intrinsic assignment for the dynamic type of that component otherwise.

The processor may perform the component-by-component assignment in any order or by any means that has the same effect.

NOTE 7.40

For an example of a derived-type intrinsic assignment statement, if C and D are of the same derived type with a pointer component P and nonpointer components S, T, U, and V of type integer, logical, character, and another derived type, respectively, the intrinsic

```
C = D
```

NOTE 7.40 (cont.)

> pointer assigns D % P to C % P. It assigns D % S to C % S, D % T to C % T, and D % U to C % U using intrinsic assignment. It assigns D % V to C % V using defined assignment if objects of that type have a compatible type-bound defined assignment, and intrinsic assignment otherwise.

NOTE 7.41

> If an allocatable component of *expr* is unallocated, the corresponding component of *variable* has an allocation status of unallocated after execution of the assignment.

When *variable* is a subobject, the assignment does not affect the definition status or value of other parts of the object. For example, if *variable* is an array section, the assignment does not affect the definition status or value of the elements of the array not specified by the array section.

7.4.1.4 Defined assignment statement

A **defined assignment statement** is an assignment statement that is defined by a subroutine and a generic interface (4.5.1, 12.3.2.1.2) that specifies ASSIGNMENT (=). A **defined elemental assignment statement** is a defined assignment statement for which the subroutine is elemental (12.7).

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

(1) The subroutine is specified with a SUBROUTINE (12.5.2.2) or ENTRY (12.5.2.4) statement that specifies two dummy arguments, d_1 and d_2,

(2) Either

  (a) A generic interface (12.3.2.1) provides the subroutine with a *generic-spec* of ASSIGNMENT (=), or

  (b) There is a generic binding (4.5.1) in the declared type of x_1 or x_2 with a *generic-spec* of ASSIGNMENT (=) and there is a corresponding binding to the subroutine in the dynamic type of x_1 or x_2, respectively,

(3) The types of d_1 and d_2 are compatible with the dynamic types of x_1 and x_2, respectively,

(4) The type parameters, if any, of d_1 and d_2 match the corresponding type parameters of x_1 and x_2, respectively, and

(5) Either

  (a) The ranks of x_1 and x_2 match those of d_1 and d_2 or

  (b) The subroutine is elemental, x_1 and x_2 are conformable, and there is no other subroutine that defines the operation.

If d_1 or d_2 is an array, the shapes of x_1 and x_2 shall match the shapes of d_1 and d_2, respectively.

7.4.1.5 Interpretation of defined assignment statements

The interpretation of a defined assignment is provided by the subroutine that defines it.

If the defined assignment is an elemental assignment and the *variable* in the assignment is an array, the defined assignment is performed element-by-element, in any order, on corresponding elements of *variable* and *expr*. If *expr* is a scalar, it 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*.

NOTE 7.42

> The rules of defined assignment (12.3.2.1.2), procedure references (12.4), subroutine references (12.4.3), and elemental subroutine arguments (12.7.3) ensure that the defined assignment has the

NOTE 7.42 (cont.)

> same effect as if the evaluation of all operations in x_2 and x_1 occurs before any portion of x_1 is defined.

7.4.2 Pointer assignment

Pointer assignment causes a pointer to become associated with a target or causes its pointer association status to become disassociated or undefined. Any previous association between the pointer and a target is broken.

Pointer assignment for a pointer component of a structure may also take place by execution of a derived-type intrinsic assignment statement (7.4.1.3).

A pointer may also become associated with a target by allocation of the pointer.

R735	*pointer-assignment-stmt*	**is**	*data-pointer-object* [(*bounds-spec-list*)] => *data-target*
		or	*data-pointer-object* (*bounds-remapping-list*) => *data-target*
		or	*proc-pointer-object* => *proc-target*
R736	*data-pointer-object*	**is**	*variable-name*
		or	*variable* % *data-pointer-component-name*

C716 (R735) If ***data-target*** is not unlimited polymorphic, ***data-pointer-object*** shall be type compatible (5.1.1.2) with it, and the corresponding kind type parameters shall be equal.

C717 (R735) If ***data-target*** is unlimited polymorphic, ***data-pointer-object*** shall be unlimited polymorphic, of a sequence derived type, or of a type with the BIND attribute.

C718 (R735) If ***bounds-spec-list*** is specified, the number of ***bounds-spec***s shall equal the rank of ***data-pointer-object***.

C719 (R735) If ***bounds-remapping-list*** is specified, the number of ***bounds-remapping***s shall equal the rank of ***data-pointer-object***.

C720 (R735) If ***bounds-remapping-list*** is specified, ***data-target*** shall have rank one; otherwise, the ranks of ***data-pointer-object*** and ***data-target*** shall be the same.

C721 (R736) A ***variable-name*** shall have the POINTER attribute.

C722 (R736) A ***data-pointer-component-name*** shall be the name of a component of ***variable*** that is a data pointer.

R737	*bounds-spec*	**is**	*lower-bound-expr* :
R738	*bounds-remapping*	**is**	*lower-bound-expr* : *upper-bound-expr*
R739	*data-target*	**is**	*variable*
		or	*expr*

C723 (R739) A ***variable*** shall have either the TARGET or POINTER attribute, and shall not be an array section with a vector subscript.

C724 (R739) An ***expr*** shall be a reference to a function whose result is a data pointer.

R740	*proc-pointer-object*	**is**	*proc-pointer-name*
		or	*proc-component-ref*
R741	*proc-component-ref*	**is**	*variable* % *procedure-component-name*

C725 (R741) the ***procedure-component-name*** shall be the name of a procedure pointer component of the declared type of ***variable***.

R742	*proc-target*	**is**	*expr*
		or	*procedure-name*
		or	*proc-component-ref*

C726 (R742) An *expr* shall be a reference to a function whose result is a procedure pointer.

C727 (R742) A *procedure-name* shall be the name of an external, module, or dummy procedure, a specific intrinsic function listed in 13.6 and not marked with a bullet (•), or a procedure pointer.

C728 (R742) The *proc-target* shall not be a nonintrinsic elemental procedure.

7.4.2.1 Data pointer assignment

If *data-pointer-object* is not polymorphic and *data-target* is polymorphic with dynamic type that differs from its declared type, the assignment target is the ancestor component of *data-target* that has the type of *data-pointer-object*. Otherwise, the assignment target is *data-target*.

If *data-target* is not a pointer, *data-pointer-object* becomes pointer associated with the assignment target. Otherwise, the pointer association status of *data-pointer-object* becomes that of *data-target*; if *data-target* is associated with an object, *data-pointer-object* becomes associated with the assignment target. If *data-target* is allocatable, it shall be allocated.

If *data-pointer-object* is polymorphic (5.1.1.2), it assumes the dynamic type of *data-target*. If *data-pointer-object* is of sequence derived type or a type with the BIND attribute, the dynamic type of *data-target* shall be that derived type.

If *data-target* is a disassociated pointer, all nondeferred type parameters of the declared type of *data-pointer-object* that correspond to nondeferred type parameters of *data-target* shall have the same values as the corresponding type parameters of *data-target*. Otherwise, all nondeferred type parameters of the declared type of *data-pointer-object* shall have the same values as the corresponding type parameters of *data-target*.

If *data-pointer-object* has nondeferred type parameters that correspond to deferred type parameters of *data-target*, *data-target* shall not be a pointer with undefined association status.

If *bounds-remapping-list* is specified, *data-target* shall not be a disassociated or undefined pointer, and the size of *data-target* shall not be less than the size of *data-pointer-object*. The elements of the target of *data-pointer-object*, in array element order (6.2.2.2), are the first SIZE(*data-pointer-object*) elements of *data-target*.

If no *bounds-remapping-list* is specified, the extent of a dimension of *data-pointer-object* is the extent of the corresponding dimension of *data-target*. If *bounds-spec-list* appears, it specifies the lower bounds; otherwise, the lower bound of each dimension is the result of the intrinsic function LBOUND (13.7.60) applied to the corresponding dimension of *data-target*. The upper bound of each dimension is one less than the sum of the lower bound and the extent.

7.4.2.2 Procedure pointer assignment

If the *proc-target* is not a pointer, *proc-pointer-object* becomes pointer associated with *proc-target*. Otherwise, the pointer association status of *proc-pointer-object* becomes that of *proc-target*; if *proc-target* is associated with a procedure, *proc-pointer-object* becomes associated with the same procedure.

If *proc-pointer-object* has an explicit interface, its characteristics shall be the same as *proc-target* except that *proc-target* may be pure even if *proc-pointer-object* is not pure and *proc-target* may be an elemental intrinsic procedure even if *proc-pointer-object* is not elemental.

If the characteristics of *proc-pointer-object* or *proc-target* are such that an explicit interface is required, both *proc-pointer-object* and *proc-target* shall have an explicit interface.

If *proc-pointer-object* has an implicit interface and is explicitly typed or referenced as a function, *proc-target* shall be a function. If *proc-pointer-object* has an implicit interface and is referenced as a subroutine, *proc-target* shall be a subroutine.

If *proc-target* and *proc-pointer-object* are functions, they shall have the same type; corresponding type parameters shall either both be deferred or both have the same value.

If *procedure-name* is a specific procedure name that is also a generic name, only the specific procedure is associated with pointer-object.

7.4.2.3 Examples

NOTE 7.43

The following are examples of pointer assignment statements. (See Note 12.14 for declarations of P and BESSEL.)

```
NEW_NODE % LEFT => CURRENT_NODE
SIMPLE_NAME => TARGET_STRUCTURE % SUBSTRUCT % COMPONENT
PTR => NULL ( )
ROW => MAT2D (N, :)
WINDOW => MAT2D (I-1:I+1, J-1:J+1)
POINTER_OBJECT => POINTER_FUNCTION (ARG_1, ARG_2)
EVERY_OTHER => VECTOR (1:N:2)
WINDOW2 (0:, 0:) => MAT2D (ML:MU, NL:NU)
! P is a procedure pointer and BESSEL is a procedure with a
! compatible interface.
P => BESSEL

! Likewise for a structure component.
STRUCT % COMPONENT => BESSEL
```

NOTE 7.44

It is possible to obtain high-rank views of (parts of) rank-one objects by specifying upper bounds in pointer assignment statements. Consider the following example, in which a matrix is under consideration. The matrix is stored as a rank-one object in MYDATA because its diagonal is needed for some reason – the diagonal cannot be gotten as a single object from a rank-two representation. The matrix is represented as a rank-two view of MYDATA.

```
real, target :: MYDATA ( NR*NC )     ! An automatic array
real, pointer :: MATRIX ( :, : )     ! A rank-two view of MYDATA
real, pointer :: VIEW_DIAG ( : )
MATRIX( 1:NR, 1:NC ) => MYDATA       ! The MATRIX view of the data
VIEW_DIAG => MYDATA( 1::NR+1 )       ! The diagonal of MATRIX
```

Rows, columns, or blocks of the matrix can be accessed as sections of MATRIX.

7.4.3 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 array expression.

7.4.3.1 General form of the masked array assignment

A **masked array assignment** is either a WHERE statement or a WHERE construct.

R743	*where-stmt*	**is**	WHERE (*mask-expr*) *where-assignment-stmt*
R744	*where-construct*	**is**	*where-construct-stmt*
			[*where-body-construct*] ...
			[*masked-elsewhere-stmt*
			[*where-body-construct*] ...] ...
			[*elsewhere-stmt*
			[*where-body-construct*] ...]
			end-where-stmt
R745	*where-construct-stmt*	**is**	[*where-construct-name*:] WHERE (*mask-expr*)
R746	*where-body-construct*	**is**	*where-assignment-stmt*
		or	*where-stmt*
		or	*where-construct*
R747	*where-assignment-stmt*	**is**	*assignment-stmt*
R748	*mask-expr*	**is**	*logical-expr*
R749	*masked-elsewhere-stmt*	**is**	ELSEWHERE (*mask-expr*) [*where-construct-name*]
R750	*elsewhere-stmt*	**is**	ELSEWHERE [*where-construct-name*]
R751	*end-where-stmt*	**is**	END WHERE [*where-construct-name*]

C729 (R747) A *where-assignment-stmt* that is a defined assignment shall be elemental.

C730 (R744) If the *where-construct-stmt* is identified by a *where-construct-name*, the corresponding *end-where-stmt* shall specify the same *where-construct-name*. If the *where-construct-stmt* is not identified by a *where-construct-name*, the corresponding *end-where-stmt* shall not specify a *where-construct-name*. If an *elsewhere-stmt* or a *masked-elsewhere-stmt* is identified by a *where-construct-name*, the corresponding *where-construct-stmt* shall specify the same *where-construct-name*.

C731 (R746) A statement that is part of a *where-body-construct* shall not be a branch target statement.

If a *where-construct* contains a *where-stmt*, a *masked-elsewhere-stmt*, or another *where-construct* then each *mask-expr* within the *where-construct* shall have the same shape. In each *where-assignment-stmt*, the *mask-expr* and the *variable* being defined shall be arrays of the same shape.

NOTE 7.45

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
ELSEWHERE
   RAINING = .TRUE.
END WHERE
```

7.4.3.2 Interpretation of masked array assignments

When a WHERE statement or a *where-construct-stmt* is executed, a control mask is established. In addition, when a WHERE construct statement is executed, a pending control mask is established. If the statement does not appear as part of a *where-body-construct*, the *mask-expr* of the statement is evaluated, and the control mask is established to be the value of *mask-expr*. The pending control mask is established to have the value .NOT. *mask-expr* upon execution of a WHERE construct statement that does not appear as part of a *where-body-construct*. The *mask-expr* is evaluated only once.

Each statement in a WHERE construct is executed in sequence.

Upon execution of a *masked-elsewhere-stmt*, the following actions take place in sequence:

(1) The control mask m_c is established to have the value of the pending control mask.
(2) The pending control mask is established to have the value m_c .AND. (.NOT. *mask-expr*).
(3) The control mask m_c is established to have the value m_c .AND. *mask-expr*.

The *mask-expr* is evaluated at most once.

Upon execution of an ELSEWHERE statement, the control mask is established to have the value of the pending control mask. No new pending control mask value is established.

Upon execution of an ENDWHERE statement, the control mask and pending control mask are established to have the values they had prior to the execution of the corresponding WHERE construct statement. Following the execution of a WHERE statement that appears as a *where-body-construct*, the control mask is established to have the value it had prior to the execution of the WHERE statement.

NOTE 7.46

The establishment of control masks and the pending control mask is illustrated with the following example:

```
WHERE(cond1)         ! Statement 1
. . .
ELSEWHERE(cond2)     ! Statement 2
. . .
ELSEWHERE            ! Statement 3
. . .
END WHERE
```

Following execution of statement 1, the control mask has the value cond1 and the pending control mask has the value .NOT. cond1. Following execution of statement 2, the control mask has the value (.NOT. cond1) .AND. cond2 and the pending control mask has the value (.NOT. cond1) .AND. (.NOT. cond2). Following execution of statement 3, the control mask has the value (.NOT. cond1) .AND. (.NOT. cond2). The false condition values are propagated through the execution of the masked ELSEWHERE statement.

Upon execution of a WHERE construct statement that is part of a *where-body-construct*, the pending control mask is established to have the value m_c .AND. (.NOT. *mask-expr*). The control mask is then established to have the value m_c .AND. *mask-expr*. The *mask-expr* is evaluated at most once.

Upon execution of a WHERE statement that is part of a *where-body-construct*, the control mask is established to have the value m_c .AND. *mask-expr*. The pending mask is not altered.

If a nonelemental function reference occurs in the *expr* or *variable* of a *where-assignment-stmt* or in a *mask-expr*, the function is evaluated without any masked control; that is, all of its argument expressions are fully evaluated and the function is fully evaluated. If the result is an array and the reference is not within the argument list of a nonelemental function, elements corresponding to true values in the control mask are selected for use in evaluating the *expr*, *variable* or *mask-expr*.

If an elemental operation or function reference occurs in the *expr* or *variable* of a *where-assignment-stmt* or in a *mask-expr*, and is not within the argument list of a nonelemental function reference, the operation is performed or the function is evaluated only for the elements corresponding to true values of the control mask.

If an array constructor appears in a *where-assignment-stmt* or in a *mask-expr*, the array constructor is evaluated without any masked control and then the *where-assignment-stmt* is executed or the *mask-expr* is evaluated.

When a *where-assignment-stmt* is executed, the values of *expr* that correspond to true values of the control mask are assigned to the corresponding elements of *variable*.

The value of the control mask is established by the execution of a WHERE statement, a WHERE construct statement, an ELSEWHERE statement, a masked ELSEWHERE statement, or an ENDWHERE statement. Subsequent changes to the value of entities in a *mask-expr* have no effect on the value of the control mask. The execution of a function reference in the mask expression of a WHERE statement is permitted to affect entities in the assignment statement.

NOTE 7.47

Examples of function references in masked array assignments are:

```
WHERE (A > 0.0)
A = LOG (A)           ! LOG is invoked only for positive elements.
A = A / SUM (LOG (A)) ! LOG is invoked for all elements
                      ! because SUM is transformational.
END WHERE
```

7.4.4 FORALL

FORALL constructs and statements are used to control the execution of assignment and pointer assignment statements with selection by sets of index values and an optional mask expression.

7.4.4.1 The FORALL Construct

The FORALL construct allows multiple assignments, masked array (WHERE) assignments, and nested FORALL constructs and statements to be controlled by a single *forall-triplet-spec-list* and *scalar-mask-expr*.

R752	*forall-construct*	**is**	*forall-construct-stmt*
			[*forall-body-construct*] ...
			end-forall-stmt
R753	*forall-construct-stmt*	**is**	[*forall-construct-name* :] FORALL *forall-header*
R754	*forall-header*	**is**	(*forall-triplet-spec-list* [, *scalar-mask-expr*])
R755	*forall-triplet-spec*	**is**	*index-name* = *subscript* : *subscript* [: *stride*]
R618	*subscript*	**is**	*scalar-int-expr*
R621	*stride*	**is**	*scalar-int-expr*
R756	*forall-body-construct*	**is**	*forall-assignment-stmt*
		or	*where-stmt*
		or	*where-construct*
		or	*forall-construct*
		or	*forall-stmt*
R757	*forall-assignment-stmt*	**is**	*assignment-stmt*
		or	*pointer-assignment-stmt*
R758	*end-forall-stmt*	**is**	END FORALL [*forall-construct-name*]

C732 (R758) If the *forall-construct-stmt* has a *forall-construct-name*, the *end-forall-stmt* shall have the same *forall-construct-name*. If the *end-forall-stmt* has a *forall-construct-name*, the *forall-*

construct-stmt shall have the same *forall-construct-name*.

C733 (R754) The *scalar-mask-expr* shall be scalar and of type logical.

C734 (R754) Any procedure referenced in the *scalar-mask-expr*, including one referenced by a defined operation, shall be a pure procedure (12.6).

C735 (R755) The *index-name* shall be a named scalar variable of type integer.

C736 (R755) A *subscript* or *stride* in a *forall-triplet-spec* shall not contain a reference to any *index-name* in the *forall-triplet-spec-list* in which it appears.

C737 (R756) A statement in a *forall-body-construct* shall not define an *index-name* of the *forall-construct*.

C738 (R756) Any procedure referenced in a *forall-body-construct*, including one referenced by a defined operation, assignment, or finalization, shall be a pure procedure.

C739 (R756) A *forall-body-construct* shall not be a branch target.

NOTE 7.48

An example of a FORALL construct is:

```
REAL :: A(10, 10), B(10, 10) = 1.0
. . .
FORALL (I = 1:10, J = 1:10, B(I, J) /= 0.0)
   A(I, J) = REAL (I + J - 2)
   B(I, J) = A(I, J) + B(I, J) * REAL (I * J)
END FORALL
```

NOTE 7.49

An assignment statement that is a FORALL body construct may be a scalar or array assignment statement, or a defined assignment statement. The variable being defined will normally use each index name in the *forall-triplet-spec-list*. For example

```
FORALL (I = 1:N, J = 1:N)
   A(:, I, :, J) = 1.0 / REAL(I + J - 1)
END FORALL
```

broadcasts scalar values to rank-two subarrays of A.

NOTE 7.50

An example of a FORALL construct containing a pointer assignment statement is:

```
TYPE ELEMENT
   REAL ELEMENT_WT
   CHARACTER (32), POINTER :: NAME
END TYPE ELEMENT
TYPE(ELEMENT) CHART(200)
REAL WEIGHTS (1000)
CHARACTER (32), TARGET :: NAMES (1000)
. . .
FORALL (I = 1:200, WEIGHTS (I + N - 1) > .5)
   CHART(I) % ELEMENT_WT = WEIGHTS (I + N - 1)
```

NOTE 7.50 (cont.)

```
      CHART(I) % NAME => NAMES (I + N - 1)
   END FORALL
```

The results of this FORALL construct cannot be achieved with a WHERE construct because a pointer assignment statement is not permitted in a WHERE construct.

An *index-name* in a *forall-construct* has a scope of the construct (16.3). It is a scalar variable that has the type and type parameters that it would have if it were the name of a variable in the scoping unit that includes the FORALL, and this type shall be integer type; it has no other attributes.

NOTE 7.51

The use of *index-name* variables in a FORALL construct does not affect variables of the same name, for example:

```
   INTEGER :: X = -1
   REAL A(5, 4)
   J = 100
   . . .
   FORALL (X = 1:5, J = 1:4)
      A (X, J) = J
   END FORALL
```

After execution of the FORALL, the variables X and J have the values -1 and 100 and A has the value

```
          1 2 3 4
          1 2 3 4
          1 2 3 4
          1 2 3 4
          1 2 3 4
```

7.4.4.2 Execution of the FORALL construct

There are three stages in the execution of a FORALL construct:

(1) Determination of the values for *index-name* variables,
(2) Evaluation of the *scalar-mask-expr*, and
(3) Execution of the FORALL body constructs.

7.4.4.2.1 Determination of the values for index variables

The subscript and stride expressions in the *forall-triplet-spec-list* are evaluated. These expressions may be evaluated in any order. The set of values that a particular *index-name* variable assumes is determined as follows:

(1) The lower bound m_1, the upper bound m_2, and the stride m_3 are of type integer with the same kind type parameter as the *index-name*. Their values are established by evaluating the first subscript, the second subscript, and the stride expressions, respectively, including, if necessary, conversion to the kind type parameter of the *index-name* according to the rules for numeric conversion (Table 7.9). If a stride does not appear, m_3 has the value 1. The value m_3 shall not be zero.
(2) Let the value of *max* be $(m_2 - m_1 + m_3)/m_3$. If *max* ≤ 0 for some *index-name*, the execution

of the construct is complete. Otherwise, the set of values for the *index-name* is

$$m_1 + (k - 1) \times m_3 \qquad \text{where } k = 1, 2, ..., \mathit{max}.$$

The set of combinations of *index-name* values is the Cartesian product of the sets defined by each triplet specification. An *index-name* becomes defined when this set is evaluated.

NOTE 7.52

The *stride* may be positive or negative; the FORALL body constructs are executed as long as *max* > 0. For the *forall-triplet-spec*

```
I = 10:1:-1
```

max has the value 10

7.4.4.2.2 Evaluation of the mask expression

The *scalar-mask-expr*, if any, is evaluated for each combination of *index-name* values. If the *scalar-mask-expr* is not present, it is as if it were present with the value true. The *index-name* variables may be primaries in the *scalar-mask-expr*.

The **active combination of *index-name* values** is defined to be the subset of all possible combinations (7.4.4.2.1) for which the *scalar-mask-expr* has the value true.

NOTE 7.53

The *index-name* variables may appear in the mask, for example

```
FORALL (I=1:10, J=1:10, A(I) > 0.0 .AND. B(J) < 1.0)
. . .
```

7.4.4.2.3 Execution of the FORALL body constructs

The *forall-body-construct*s are executed in the order in which they appear. Each construct is executed for all active combinations of the *index-name* values with the following interpretation:

Execution of a *forall-assignment-stmt* that is an *assignment-stmt* causes the evaluation of *expr* and all expressions within *variable* for all active combinations of *index-name* values. These evaluations may be done in any order. After all these evaluations have been performed, each *expr* value is assigned to the corresponding *variable*. The assignments may occur in any order.

Execution of a *forall-assignment-stmt* that is a *pointer-assignment-stmt* causes the evaluation of all expressions within *data-target* and *data-pointer-object* or *proc-target* and *proc-pointer-object*, the determination of any pointers within *data-pointer-object* or *proc-pointer-object*, and the determination of the target for all active combinations of *index-name* values. These evaluations may be done in any order. After all these evaluations have been performed, each *data-pointer-object* or *proc-pointer-object* is associated with the corresponding target. These associations may occur in any order.

In a *forall-assignment-stmt*, a defined assignment subroutine shall not reference any *variable* that becomes defined by the statement.

NOTE 7.54

The following FORALL construct contains two assignment statements. The assignment to array B uses the values of array A computed in the previous statement, not the values A had prior to execution of the FORALL.

NOTE 7.54 (cont.)

```
FORALL (I = 2:N-1, J = 2:N-1 )
   A (I, J) = A(I, J-1) + A(I, J+1) + A(I-1, J) + A(I+1, J)
   B (I, J) = 1.0 / A(I, J)
END FORALL
```

Computations that would otherwise cause error conditions can be avoided by using an appropriate *scalar-mask-expr* that limits the active combinations of the *index-name* values. For example:

```
FORALL (I = 1:N, Y(I) /= 0.0)
   X(I) = 1.0 / Y(I)
END FORALL
```

Each statement in a *where-construct* (7.4.3) within a *forall-construct* is executed in sequence. When a *where-stmt*, *where-construct-stmt* or *masked-elsewhere-stmt* is executed, the statement's *mask-expr* is evaluated for all active combinations of *index-name* values as determined by the outer *forall-construct*s, masked by any control mask corresponding to outer *where-construct*s. Any *where-assignment-stmt* is executed for all active combinations of *index-name* values, masked by the control mask in effect for the *where-assignment-stmt*.

NOTE 7.55

This FORALL construct contains a WHERE statement and an assignment statement.

```
INTEGER A(5,4), B(5,4)
FORALL ( I = 1:5 )
   WHERE ( A(I,:) == 0 ) A(I,:) = I
   B (I,:) = I / A(I,:)
END FORALL
```

When executed with the input array

```
        0  0  0  0
        1  1  1  0
A  =    2  2  0  2
        1  0  2  3
        0  0  0  0
```

the results will be

```
        1  1  1  1              1  1  1  1
        1  1  1  2              2  2  2  1
A  =    2  2  3  2       B  =   1  1  1  1
        1  4  2  3              4  1  2  1
        5  5  5  5              1  1  1  1
```

For an example of a FORALL construct containing a WHERE construct with an ELSEWHERE statement, see C.4.5.

Execution of a *forall-stmt* or *forall-construct* causes the evaluation of the *subscript* and *stride* expressions in the *forall-triplet-spec-list* for all active combinations of the *index-name* values of the outer FORALL construct. The set of combinations of *index-name* values for the inner FORALL is the union of the sets defined by these bounds and strides for each active combination of the outer *index-name* values; it also includes the outer *index-name* values. The *scalar-mask-expr* is then evaluated for all combinations of the *index-name* values of the inner construct to produce a set of active combinations for the inner construct. If there is no *scalar-mask-expr*, it is as if it were present with the value .TRUE.. Each statement in the inner FORALL is then executed for each active combination of the *index-name* values.

NOTE 7.56

This FORALL construct contains a nested FORALL construct. It assigns the transpose of the strict lower triangle of array A (the section below the main diagonal) to the strict upper triangle of A.

```
INTEGER A (3, 3)
FORALL (I = 1:N-1 )
   FORALL ( J=I+1:N )
      A(I,J) = A(J,I)
   END FORALL
END FORALL
```

If prior to execution N = 3 and

```
        0  3  6
A  =    1  4  7
        2  5  8
```

then after execution

```
        0  1  2
A  =    1  4  5
        2  5  8
```

7.4.4.3 The FORALL statement

The FORALL statement allows a single assignment statement or pointer assignment to be controlled by a set of index values and an optional mask expression.

R759 *forall-stmt* **is** FORALL *forall-header forall-assignment-stmt*

A FORALL statement is equivalent to a FORALL construct containing a single *forall-body-construct* that is a *forall-assignment-stmt*.

The scope of an *index-name* in a *forall-stmt* is the statement itself (16.3).

NOTE 7.57

Examples of FORALL statements are:

```
FORALL (I=1:N) A(I,I) = X(I)
```

This statement assigns the elements of vector X to the elements of the main diagonal of matrix A.

```
FORALL (I = 1:N, J = 1:N)  X(I,J) = 1.0 / REAL (I+J-1)
```

Array element X(I,J) is assigned the value (1.0 / REAL (I+J-1)) for values of I and J between 1 and N, inclusive.

NOTE 7.57 (cont.)

```
FORALL (I=1:N, J=1:N, Y(I,J) /= 0 .AND. I /= J)  X(I,J) = 1.0 / Y(I,J)
```

This statement takes the reciprocal of each nonzero off-diagonal element of array Y(1:N, 1:N) and assigns it to the corresponding element of array X. Elements of Y that are zero or on the diagonal do not participate, and no assignments are made to the corresponding elements of X. The results from the execution of the example in Note 7.56 could be obtained with a single FORALL statement:

```
FORALL ( I = 1:N-1, J=1:N, J > I )  A(I,J) = A(J,I)
```

For more examples of FORALL statements, see C.4.6.

7.4.4.4 Restrictions on FORALL constructs and statements

A many-to-one assignment is more than one assignment to the same object, or association of more than one target with the same pointer, whether the object is referenced directly or indirectly through a pointer. A many-to-one assignment shall not occur within a single statement in a FORALL construct or statement. It is possible to assign or pointer assign to the same object in different assignment statements in a FORALL construct.

NOTE 7.58

The appearance of each *index-name* in the identification of the left-hand side of an assignment statement is helpful in eliminating many-to-one assignments, but it is not sufficient to guarantee there will be none. For example, the following is allowed

```
FORALL (I = 1:10)
   A (INDEX (I)) = B(I)
END FORALL
```

if and only if INDEX(1:10) contains no repeated values.

Within the scope of a FORALL construct, a nested FORALL statement or FORALL construct shall not have the same *index-name*. The *forall-header* expressions within a nested FORALL may depend on the values of outer *index-name* variables.

Section 8: Execution control

The execution sequence may be controlled by constructs containing blocks and by certain executable statements that are used to alter the execution sequence.

8.1 Executable constructs containing blocks

The following are executable constructs that contain blocks:

(1) ASSOCIATE construct
(2) CASE construct
(3) DO construct
(4) IF construct
(5) SELECT TYPE construct

There is also a nonblock form of the DO construct.

A **block** is a sequence of executable constructs that is treated as a unit.

R801 *block* **is** [*execution-part-construct*] ...

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; however, in some forms of the DO construct, a sequence of executable constructs without a terminating boundary statement shall obey all other rules governing blocks (8.1.1).

NOTE 8.1

A block need not contain any executable constructs. Execution of such a block has no effect.

Any of these constructs may be named. If a construct is named, the name shall be the first lexical token of the first statement of the construct and the last lexical token of the construct. In fixed source form, the name preceding the construct shall be placed after character position 6.

A statement **belongs** to the innermost construct in which it appears unless it contains a construct name, in which case it belongs to the named construct.

NOTE 8.2

An example of a construct containing a block is:

```
IF (A > 0.0) THEN
   B = SQRT (A)  ! These two statements
   C = LOG (A)   ! form a block.
END IF
```

8.1.1 Rules governing blocks

8.1.1.1 Executable constructs in blocks

If a block contains an executable construct, the executable construct shall be entirely 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 and transfers from the interior of a block to outside the block may occur.

Subroutine and function references (12.4.2, 12.4.3) may appear in a block.

8.1.1.3 Execution of a block

Execution of a block begins with the execution of the first executable construct in the block. Execution of the block is completed when the last executable construct in the sequence is executed or when a branch out of the block takes place.

NOTE 8.3

The action that takes place at the terminal boundary depends on the particular construct and on the block within that construct. It is usually a transfer of control.

8.1.2 IF construct

The **IF construct** selects for execution at most one of its constituent blocks. The selection is based on a sequence of logical expressions. The **IF statement** controls the execution of a single statement (8.1.2.4) based on a single logical expression.

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-logical-expr ) THEN
R804  else-if-stmt      is  ELSE IF ( scalar-logical-expr ) THEN [ if-construct-name ]
R805  else-stmt         is  ELSE [ if-construct-name ]
R806  end-if-stmt       is  END IF [ if-construct-name ]
```

C801 (R802) If the *if-then-stmt* of an *if-construct* specifies an *if-construct-name*, the corresponding *end-if-stmt* shall specify the same *if-construct-name*. If the *if-then-stmt* of an *if-construct* does not specify an *if-construct-name*, the corresponding *end-if-stmt* shall not specify an *if-construct-name*. If an *else-if-stmt* or *else-stmt* specifies an *if-construct-name*, the corresponding *if-then-stmt* shall specify the same *if-construct-name*.

8.1.2.2 Execution of an IF construct

At most one of the blocks in the IF construct is executed. If there is an ELSE statement in the construct, exactly one of the blocks in the construct is executed. The scalar logical 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 scalar logical expressions in any remaining ELSE IF statements of the IF construct are not evaluated. If none of the evaluated expressions is true and there is no ELSE statement, the execution of the construct is completed without the execution of any block within the construct.

An ELSE IF statement or an ELSE statement shall not be a branch target statement. It is permissible to branch to an END IF statement only from within its IF construct. Execution of an END IF statement has no effect.

8.1.2.3 Examples of IF constructs

NOTE 8.4

```
IF (CVAR == 'RESET') THEN
   I = 0; J = 0; K = 0
END IF
PROOF_DONE: IF (PROP) THEN
   WRITE (3, '(''QED'')')
   STOP
ELSE
   PROP = NEXTPROP
END IF PROOF_DONE
IF (A > 0) THEN
   B = C/A
   IF (B > 0) THEN
      D = 1.0
   END IF
ELSE IF (C > 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 (R214).

R807	*if-stmt*	**is**	IF (*scalar-logical-expr*) *action-stmt*

C802 (R807) The *action-stmt* in the *if-stmt* shall not be an *if-stmt*, *end-program-stmt*, *end-function-stmt*, or *end-subroutine-stmt*.

Execution of an IF statement causes evaluation of the scalar logical 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.

The execution of a function reference in the scalar logical expression may affect entities in the action statement.

NOTE 8.5

An example of an IF statement is:

```
IF (A > 0.0) A = LOG (A)
```

8.1.3 CASE construct

The **CASE construct** selects for execution at most one of its constituent blocks. The selection is based on the value of an expression.

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**	[*case-construct-name* :] SELECT CASE (*case-expr*)
R810	*case-stmt*	**is**	CASE *case-selector* [*case-construct-name*]
R811	*end-select-stmt*	**is**	END SELECT [*case-construct-name*]

C803 (R808) If the *select-case-stmt* of a *case-construct* specifies a *case-construct-name*, the corresponding *end-select-stmt* shall specify the same *case-construct-name*. If the *select-case-stmt* of a *case-construct* does not specify a *case-construct-name*, the corresponding *end-select-stmt* shall not specify a *case-construct-name*. If a *case-stmt* specifies a *case-construct-name*, the corresponding *select-case-stmt* shall specify the same *case-construct-name*.

R812	*case-expr*	**is**	*scalar-int-expr*
		or	*scalar-char-expr*
		or	*scalar-logical-expr*
R813	*case-selector*	**is**	(*case-value-range-list*)
		or	DEFAULT

C804 (R808) No more than one of the selectors of one of the CASE statements shall be DEFAULT.

R814	*case-value-range*	**is**	*case-value*
		or	*case-value* :
		or	: *case-value*
		or	*case-value* : *case-value*
R815	*case-value*	**is**	*scalar-int-initialization-expr*
		or	*scalar-char-initialization-expr*
		or	*scalar-logical-initialization-expr*

C805 (R808) For a given *case-construct*, each *case-value* shall be of the same type as *case-expr*. For character type, the kind type parameters shall be the same; character length differences are allowed.

C806 (R808) A *case-value-range* using a colon shall not be used if *case-expr* is of type logical.

C807 (R808) For a given *case-construct*, the *case-value-range*s shall not overlap; that is, there shall be no possible value of the *case-expr* that matches more than one *case-value-range*.

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**. 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 type logical if the expression c .EQV. v is true, and a match occurs for type integer or character if the expression $c == v$ is true.

(2) If the case value range is of the form *low* : *high*, a match occurs if the expression *low* <= c .AND. c <= *high* is true.

(3) If the case value range is of the form *low* :, a match occurs if the expression *low* <= c is true.

(4) If the case value range is of the form : *high*, a match occurs if the expression c <= *high* is true.

(5) If no other selector matches and a DEFAULT selector appears, it matches the case index.

(6) If no other selector matches and the DEFAULT selector does not appear, there is no match.

The block following the CASE statement containing the matching selector, if any, is executed. This completes execution of the construct.

At most one of the blocks of a CASE construct is executed.

A CASE statement shall not be a branch target statement. It is permissible to branch to an *end-select-stmt* only from within its CASE construct.

8.1.3.3 Examples of CASE constructs

NOTE 8.6

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
```

NOTE 8.7

A code fragment to check for balanced parentheses:

```
CHARACTER (80) :: LINE
   ...
LEVEL = 0
SCAN_LINE: DO I = 1, 80
   CHECK_PARENS: SELECT CASE (LINE (I:I))
   CASE ('(')
      LEVEL = LEVEL + 1
   CASE (')')
      LEVEL = LEVEL - 1
      IF (LEVEL < 0) THEN
         PRINT *, 'UNEXPECTED RIGHT PARENTHESIS'
         EXIT SCAN_LINE
      END IF
   CASE DEFAULT
      ! Ignore all other characters
   END SELECT CHECK_PARENS
END DO SCAN_LINE
IF (LEVEL > 0) THEN
   PRINT *, 'MISSING RIGHT PARENTHESIS'
END IF
```

NOTE 8.8

The following three fragments are equivalent:

```
IF (SILLY == 1) THEN
   CALL THIS
ELSE
   CALL THAT
END IF
SELECT CASE (SILLY == 1)
CASE (.TRUE.)
   CALL THIS
CASE (.FALSE.)
   CALL THAT
END SELECT
SELECT CASE (SILLY)
CASE DEFAULT
   CALL THAT
CASE (1)
   CALL THIS
END SELECT
```

NOTE 8.9

A code fragment showing several selections of one block:

```
SELECT CASE (N)
CASE (1, 3:5, 8)  ! Selects 1, 3, 4, 5, 8
   CALL SUB
CASE DEFAULT
   CALL OTHER
END SELECT
```

8.1.4 ASSOCIATE construct

The ASSOCIATE construct associates named entities with expressions or variables during the execution of its block. These named construct entities (16.3) are associating entities (16.4.1.5). The names are **associate names**.

8.1.4.1 Form of the ASSOCIATE construct

R816	*associate-construct*	**is**	*associate-stmt*
			block
			end-associate-stmt
R817	*associate-stmt*	**is**	[*associate-construct-name* :] ASSOCIATE ■
			■ (*association-list*)
R818	*association*	**is**	*associate-name* => *selector*
R819	*selector*	**is**	*expr*
		or	*variable*

C808 (R818) If *selector* is not a *variable* or is a *variable* that has a vector subscript, *associate-name* shall not appear in a variable definition context (16.5.7).

C809 (R818) An *associate-name* shall not be the same as another *associate-name* in the same *associate-stmt*.

R820 *end-associate-stmt* **is** END ASSOCIATE [*associate-construct-name*]

C810 (R820) If the *associate-stmt* of an *associate-construct* specifies an *associate-construct-name*, the corresponding *end-associate-stmt* shall specify the same *associate-construct-name*. If the *associate-stmt* of an *associate-construct* does not specify an *associate-construct-name*, the corresponding *end-associate-stmt* shall not specify an *associate-construct-name*.

8.1.4.2 Execution of the ASSOCIATE construct

Execution of an ASSOCIATE construct causes execution of its *associate-stmt* followed by execution of its block. During execution of that block each associate name identifies an entity, which is associated (16.4.1.5) with the corresponding selector. The associating entity assumes the declared type and type parameters of the selector. If and only if the selector is polymorphic, the associating entity is polymorphic.

The other attributes of the associating entity are described in 8.1.4.3.

It is permissible to branch to an *end-associate-stmt* only from within its ASSOCIATE construct.

8.1.4.3 Attributes of associate names

Within a SELECT TYPE or ASSOCIATE construct, each associating entity has the same rank as its associated selector. The lower bound of each dimension is the result of the intrinsic function LBOUND (13.7.60) applied to the corresponding dimension of *selector*. The upper bound of each dimension is one less than the sum of the lower bound and the extent. The associating entity has the ASYNCHRONOUS, INTENT, TARGET, or VOLATILE attribute if and only if the selector is a variable and has the attribute. If the associating entity is polymorphic, it assumes the dynamic type and type parameter values of the selector. If the selector has the OPTIONAL attribute, it shall be present.

If the selector (8.1.4.1) is not permitted to appear in a variable definition context (16.5.7) or is an array with a vector subscript, the associate name shall not appear in a variable definition context.

8.1.4.4 Examples of the ASSOCIATE construct

NOTE 8.10

The following example illustrates an association with an expression.

```
ASSOCIATE ( Z => EXP(-(X**2+Y**2)) * COS(THETA) )
  PRINT *, A+Z, A-Z
END ASSOCIATE
```

The following example illustrates an association with a derived-type variable.

```
ASSOCIATE ( XC => AX%B(I,J)%C )
  XC%DV = XC%DV + PRODUCT(XC%EV(1:N))
END ASSOCIATE
```

The following example illustrates association with an array section.

```
ASSOCIATE ( ARRAY => AX%B(I,:)%C )
  ARRAY(N)%EV = ARRAY(N-1)%EV
END ASSOCIATE
```

The following example illustrates multiple associations.

NOTE 8.10 (cont.)

```
ASSOCIATE ( W => RESULT(I,J)%W, ZX => AX%B(I,J)%D, ZY => AY%B(I,J)%D )
  W = ZX*X + ZY*Y
END ASSOCIATE
```

8.1.5 SELECT TYPE construct

The SELECT TYPE construct selects for execution at most one of its constituent blocks. The selection is based on the dynamic type of an expression. A name is associated with the expression (16.3, 16.4.1.5), in the same way as for the ASSOCIATE construct.

8.1.5.1 Form of the SELECT TYPE construct

R821 *select-type-construct* **is** *select-type-stmt*
  [*type-guard-stmt*
    block] ...
  end-select-type-stmt

R822 *select-type-stmt* **is** [*select-construct-name* :] SELECT TYPE ■
  ■ ([*associate-name* =>] *selector*)

C811 (R822) If *selector* is not a named *variable*, *associate-name* => shall appear.

C812 (R822) If *selector* is not a *variable* or is a *variable* that has a vector subscript, *associate-name* shall not appear in a variable definition context (16.5.7).

C813 (R822) The *selector* in a *select-type-stmt* shall be polymorphic.

R823 *type-guard-stmt* **is** TYPE IS (*type-spec*) [*select-construct-name*]
  or CLASS IS (*type-spec*) [*select-construct-name*]
  or CLASS DEFAULT [*select-construct-name*]

C814 (R823) The *type-spec* shall specify that each length type parameter is assumed.

C815 (R823) The *type-spec* shall not specify a sequence derived type or a type with the BIND attribute.

C816 (R823) If *selector* is not unlimited polymorphic, the *type-spec* shall specify an extension of the declared type of *selector*.

C817 (R823) For a given *select-type-construct*, the same type and kind type parameter values shall not be specified in more than one TYPE IS *type-guard-stmt* and shall not be specified in more than one CLASS IS *type-guard-stmt*.

C818 (R823) For a given *select-type-construct*, t shall *type-* be test one CLASSEFAULT *guard-stmt*

R843 *end-select-type-stmt* **is** SELE(T)-334([)]TJ/F299.963Tf74.30530Td[(sel)1(e)51(ct-c)51(onstruct-name)]T

C898 (R813) If the select-type-stmt
sponnn shall *end-select-type-stmt* *select-construct-name* If the
type-(t)-1(m)1(t)]TJ/F8 9 Tf 43.290 Td If a
shall *select-type-stmt* ea
thesp, *select-construct-name* The
.

The *associate-name* of a ECT TYPE onstruct ie
thatons(it)1uteise *selector*.

8.1.5.2 Execution of the SELECT TYPE construct

Execution of a SELECT TYPE construct whose selector is not a *variable* causes the selector expression to be evaluated.

A SELECT TYPE construct selects at most one block to be executed. During execution of that block, the associate name identifies an entity, which is associated (16.4.1.5) with the selector.

A TYPE IS type guard statement matches the selector if the dynamic type and type parameter values of the selector are the same as those specified by the statement. A CLASS IS type guard statement matches the selector if the dynamic type of the selector is an extension of the type specified by the statement and the kind type parameter values specified by the statement are the same as the corresponding type parameter values of the dynamic type of the selector.

The block to be executed is selected as follows:

(1) If a TYPE IS type guard statement matches the selector, the block following that statement is executed.

(2) Otherwise, if exactly one CLASS IS type guard statement matches the selector, the block following that statement is executed.

(3) Otherwise, if several CLASS IS type guard statements match the selector, one of these statements must specify a type that is an extension of all the types specified in the others; the block following that statement is executed.

(4) Otherwise, if there is a CLASS DEFAULT type guard statement, the block following that statement is executed.

NOTE 8.11

This algorithm does not examine the type guard statements in source text order when it looks for a match; it selects the most particular type guard when there are several potential matches.

Within the block following a TYPE IS type guard statement, the associating entity (16.4.5) is not polymorphic (5.1.1.2), has the type named in the type guard statement, and has the type parameter values of the selector.

Within the block following a CLASS IS type guard statement, the associating entity is polymorphic and has the declared type named in the type guard statement. The type parameter values of the associating entity are the corresponding type parameter values of the selector.

Within the block following a CLASS DEFAULT type guard statement, the associating entity is polymorphic and has the same declared type as the selector. The type parameter values of the associating entity are those of the declared type of the selector.

NOTE 8.12

If the declared type of the *selector* is T, specifying CLASS DEFAULT has the same effect as specifying CLASS IS (T).

The other attributes of the associating entity are described in 8.1.4.3.

A type guard statement shall not be a branch target statement. It is permissible to branch to an *end-select-type-stmt* only from within its SELECT TYPE construct.

8.1.5.3 Examples of the SELECT TYPE construct

NOTE 8.13

```
TYPE POINT
  REAL :: X, Y
END TYPE POINT
TYPE, EXTENDS(POINT) :: POINT_3D
  REAL :: Z
END TYPE POINT_3D
TYPE, EXTENDS(POINT) :: COLOR_POINT
  INTEGER :: COLOR
END TYPE COLOR_POINT

TYPE(POINT), TARGET :: P
TYPE(POINT_3D), TARGET :: P3
TYPE(COLOR_POINT), TARGET :: C
CLASS(POINT), POINTER :: P_OR_C
P_OR_C => C
SELECT TYPE ( A => P_OR_C )
CLASS IS ( POINT )
  ! "CLASS ( POINT ) :: A" implied here
  PRINT *, A%X, A%Y ! This block gets executed
TYPE IS ( POINT_3D )
  ! "TYPE ( POINT_3D ) :: A" implied here
  PRINT *, A%X, A%Y, A%Z
END SELECT
```

NOTE 8.14

The following example illustrates the omission of *associate-name*. It uses the declarations from Note 8.13.

```
P_OR_C => P3
SELECT TYPE ( P_OR_C )
CLASS IS ( POINT )
  ! "CLASS ( POINT ) :: P_OR_C" implied here
  PRINT *, P_OR_C%X, P_OR_C%Y
TYPE IS ( POINT_3D )
  ! "TYPE ( POINT_3D ) :: P_OR_C" implied here
  PRINT *, P_OR_C%X, P_OR_C%Y, P_OR_C%Z  ! This block gets executed
END SELECT
```

8.1.6 DO construct

The DO construct specifies the repeated execution of a sequence of executable constructs. Such a repeated sequence is called a **loop**. The EXIT and CYCLE statements may be used to modify the execution of a loop.

The number of iterations of a loop may be determined at the beginning of execution of the DO construct, or may be left indefinite ("DO forever" or DO WHILE). In either case, an EXIT statement (8.1.6.4.4) anywhere in the DO construct may be executed to terminate the loop immediately. The current iteration of the loop may be curtailed by executing a CYCLE statement (8.1.6.4.3).

8.1.6.1 Forms of the DO construct

The DO construct may be written in either a block form or a nonblock form.

R825	*do-construct*	**is**	*block-do-construct*
		or	*nonblock-do-construct*

8.1.6.1.1 Form of the block DO construct

R826	*block-do-construct*	**is**	*do-stmt* *do-block* *end-do*
R827	*do-stmt*	**is**	*label-do-stmt*
		or	*nonlabel-do-stmt*
R828	*label-do-stmt*	**is**	[*do-construct-name* :] DO *label* [*loop-control*]
R829	*nonlabel-do-stmt*	**is**	[*do-construct-name* :] DO [*loop-control*]
R830	*loop-control*	**is**	[,] *do-variable* = *scalar-int-expr*, *scalar-int-expr* ■ ■ [, *scalar-int-expr*]
		or	[,] WHILE (*scalar-logical-expr*)
R831	*do-variable*	**is**	*scalar-int-variable*

C820 (R831) The *do-variable* shall be a named scalar variable of type integer.

R832	*do-block*	**is**	*block*
R833	*end-do*	**is**	*end-do-stmt*
		or	*continue-stmt*
R834	*end-do-stmt*	**is**	END DO [*do-construct-name*]

C821 (R826) If the *do-stmt* of a *block-do-construct* specifies a *do-construct-name*, the corresponding *end-do* shall be an *end-do-stmt* specifying the same *do-construct-name*. If the *do-stmt* of a *block-do-construct* does not specify a *do-construct-name*, the corresponding *end-do* shall not specify a *do-construct-name*.

C822 (R826) If the *do-stmt* is a *nonlabel-do-stmt*, the corresponding *end-do* shall be an *end-do-stmt*.

C823 (R826) If the *do-stmt* is a *label-do-stmt*, the corresponding *end-do* shall be identified with the same *label*.

8.1.6.1.2 Form of the nonblock DO construct

R835	*nonblock-do-construct*	**is**	*action-term-do-construct*
		or	*outer-shared-do-construct*
R836	*action-term-do-construct*	**is**	*label-do-stmt* *do-body* *do-term-action-stmt*
R837	*do-body*	**is**	[*execution-part-construct*] ...
R838	*do-term-action-stmt*	**is**	action-stmt

C824 (R838) A *do-term-action-stmt* shall not be a *continue-stmt*, a *goto-stmt*, a *return-stmt*, a *stop-stmt*, an *exit-stmt*, a *cycle-stmt*, an *end-function-stmt*, an *end-subroutine-stmt*, an *end-program-stmt*, or an *arithmetic-if-stmt*.

C825 (R835) The *do-term-action-stmt* shall be identified with a label and the corresponding *label-do-stmt* shall refer to the same label.

R839	*outer-shared-do-construct*	**is**	*label-do-stmt* *do-body* *shared-term-do-construct*
R840	*shared-term-do-construct*	**is**	*outer-shared-do-construct*
		or	*inner-shared-do-construct*
R841	*inner-shared-do-construct*	**is**	*label-do-stmt*

do-body
do-term-shared-stmt

R842 *do-term-shared-stmt* **is** action-stmt

C826 (R842) A *do-term-shared-stmt* shall not be a *goto-stmt*, a *return-stmt*, a *stop-stmt*, an *exit-stmt*, a *cycle-stmt*, an *end-function-stmt*, an *end-subroutine-stmt*, an *end-program-stmt*, or an *arithmetic-if-stmt*.

C827 (R840) The *do-term-shared-stmt* shall be identified with a label and all of the *label-do-stmt*s of the *inner-shared-do-construct* and *outer-shared-do-construct* shall refer to the same label.

The *do-term-action-stmt*, *do-term-shared-stmt*, or *shared-term-do-construct* following the *do-body* of a nonblock DO construct is called the **DO termination** of that construct.

Within a scoping unit, all DO constructs whose DO statements refer to the same label are nonblock DO constructs, and are said to share the statement identified by that label.

8.1.6.2 Range of the DO construct

The **range** of a block DO construct is the *do-block*, which shall satisfy the rules for blocks (8.1.1). In particular, transfer of control to the interior of such a block from outside the block is prohibited. It is permitted to branch to the *end-do* of a block DO construct only from within the range of that DO construct.

The range of a nonblock DO construct consists of the *do-body* and the following DO termination. The end of such a range is not bounded by a particular statement as for the other executable constructs (e.g., END IF); nevertheless, the range satisfies the rules for blocks (8.1.1). Transfer of control into the *do-body* or to the DO termination from outside the range is prohibited; in particular, it is permitted to branch to a *do-term-shared-stmt* only from within the range of the corresponding *inner-shared-do-construct*.

8.1.6.3 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 it terminates (8.1.6.4.4).

8.1.6.4 Execution of a DO construct

A DO construct specifies a loop, that 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 range, and termination of the loop.

8.1.6.4.1 Loop initiation

When the DO statement is executed, the DO construct becomes active. If *loop-control* is

[,] *do-variable* = *scalar-int-expr* $_1$, *scalar-int-expr* $_2$ [, *scalar-int-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 of type integer with the same kind type parameter as the *do-variable*. Their values are established by evaluating *scalar-int-expr* $_1$, *scalar-int-expr* $_2$, and *scalar-int-expr* $_3$, respectively, including, if necessary, conversion to the kind type parameter of the *do-variable* according to the rules for numeric conversion (Table 7.9). If *scalar-int-expr* $_3$ does not appear, m_3 has the value 1. The value of m_3 shall not be 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 $(m_2 - m_1 + m_3)/m_3$, unless that value is negative, in which case the iteration count is 0.

NOTE 8.15

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* is omitted, no iteration count is calculated. The effect is as if a large positive iteration count, impossible to decrement to zero, were established. If *loop-control* is [,] WHILE (*scalar-logical-expr*), the effect is as if *loop-control* were omitted and the following statement inserted as the first statement of the *do-block*:

```
IF (.NOT. (scalar-logical-expr)) EXIT
```

At the completion of the execution of the DO statement, the execution cycle begins.

8.1.6.4.2 The execution cycle

The **execution cycle** of a DO construct consists of the following steps performed in sequence repeatedly until termination:

(1) The iteration count, if any, is tested. If it is zero, the loop terminates and the DO construct becomes inactive. If *loop-control* is [,] WHILE (*scalar-logical-expr*), the *scalar-logical-expr* is evaluated; if the value of this expression is false, the loop terminates and the DO construct becomes inactive. If, as a result, all of the DO constructs sharing the *do-term-shared-stmt* are inactive, the execution of all of these constructs is complete. However, if some of the DO constructs sharing the *do-term-shared-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 loop 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.

Except for the incrementation of the DO variable that occurs in step (3), the DO variable shall neither be redefined nor become undefined while the DO construct is active.

8.1.6.4.3 CYCLE statement

Step (2) in the above execution cycle may be curtailed by executing a CYCLE statement from within the range of the loop.

R843 *cycle-stmt* **is** CYCLE [*do-construct-name*]

C828 (R843) If a *cycle-stmt* refers to a *do-construct-name*, it shall be within the range of that *do-construct*; otherwise, it shall be within the range of at least one *do-construct*.

A CYCLE statement belongs to a particular DO construct. If the CYCLE statement refers to a DO construct name, it belongs to that DO construct; otherwise, it belongs to the innermost DO construct in which it appears.

Execution of a CYCLE statement causes immediate progression to step (3) of the current execution cycle of the DO construct to which it belongs. If this construct is a nonblock DO construct, the *do-term-action-stmt* or *do-term-shared-stmt* is not executed.

In a block DO construct, a transfer of control to the *end-do* has the same effect as execution of a CYCLE statement belonging to that construct. In a nonblock DO construct, transfer of control to the *do-term-action-stmt* or *do-term-shared-stmt* causes that statement or construct itself to be executed. Unless a further transfer of control results, step (3) of the current execution cycle of the DO construct is then executed.

8.1.6.4.4 Loop termination

The EXIT statement provides one way of terminating a loop.

R844 *exit-stmt* **is** EXIT [*do-construct-name*]

C829 (R844) If an *exit-stmt* refers to a *do-construct-name*, it shall be within the range of that *do-construct*; otherwise, it shall be within the range of at least one *do-construct*.

An EXIT statement belongs to a particular DO construct. If the EXIT statement refers to a DO construct name, it belongs to that DO construct; otherwise, it belongs to the innermost DO construct in which it appears.

The loop terminates, and the DO construct becomes inactive, when any of the following occurs:

(1) Determination that the iteration count is zero or the *scalar-logical-expr* is false, when tested during step (1) of the above execution cycle
(2) Execution of an EXIT statement belonging to the DO construct
(3) Execution of an EXIT statement or a CYCLE statement that is within the range of the DO construct, but that belongs to an outer DO construct
(4) Transfer of control from a statement within the range of a DO construct to a statement that is neither the *end-do* nor within the range of the same DO construct
(5) Execution of a RETURN statement within the range of the DO construct
(6) Execution of a STOP statement anywhere in the program; or termination of the program for any other reason.

When a DO construct becomes inactive, the DO variable, if any, of the DO construct retains its last defined value.

8.1.6.5 Examples of DO constructs

NOTE 8.16

The following program fragment computes a tensor product of two arrays:

```
DO I = 1, M
   DO J = 1, N
      C (I, J) = SUM (A (I, J, :) * B (:, I, J))
   END DO
END DO
```

NOTE 8.17

The following program fragment contains a DO construct that uses the WHILE form of *loop-control*. The loop will continue to execute until an end-of-file or input/output error is encountered, at which point the DO statement terminates the loop. When a negative value of X is read, the program skips immediately to the next READ statement, bypassing most of the range of the loop.

```
READ (IUN, '(1X, G14.7)', IOSTAT = IOS) X
DO WHILE (IOS == 0)
   IF (X >= 0.) THEN
      CALL SUBA (X)
```

NOTE 8.17 (cont.)

```
      CALL SUBB (X)
           ...
      CALL SUBZ (X)
   ENDIF
   READ (IUN, '(1X, G14.7)', IOSTAT = IOS) X
END DO
```

NOTE 8.18

The following example behaves exactly the same as the one in Note 8.17. However, the READ statement has been moved to the interior of the range, so that only one READ statement is needed. Also, a CYCLE statement has been used to avoid an extra level of IF nesting.

```
DO     ! A "DO WHILE + 1/2" loop
   READ (IUN, '(1X, G14.7)', IOSTAT = IOS) X
   IF (IOS /= 0) EXIT
   IF (X < 0.) CYCLE
   CALL SUBA (X)
   CALL SUBB (X)
      . . .
   CALL SUBZ (X)
END DO
```

NOTE 8.19

Additional examples of DO constructs are in C.5.3.

8.2 Branching

Branching is used to alter the normal execution sequence. A branch causes a transfer of control from one statement in a scoping unit to a labeled branch target statement in the same scoping unit. Branching may be caused by a GOTO statement, a computed GOTO statement, an arithmetic IF statement, a CALL statement that has an *alt-return-spec*, or an input/output statement that has an END= or ERR= specifier. Although procedure references and control constructs can cause transfer of control, they are not branches. A **branch target statement** is an *action-stmt*, an *associate-stmt*, an *end-associate-stmt*, an *if-then-stmt*, an *end-if-stmt*, a *select-case-stmt*, an *end-select-stmt*, a *select-type-stmt*, an *end-select-type-stmt*, a *do-stmt*, an *end-do-stmt*, a *forall-construct-stmt*, a *do-term-action-stmt*, a *do-term-shared-stmt*, or a *where-construct-stmt*.

8.2.1 GO TO statement

R845 *goto-stmt* **is** GO TO *label*

C830 (R845) The *label* shall be the statement label of a branch target statement that appears in the same scoping unit as the *goto-stmt*.

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

8.2.2 Computed GO TO statement

R846 *computed-goto-stmt* **is** GO TO (*label-list*) [,] *scalar-int-expr*

C831 (R846 Each *label* in *label-list* shall be the statement label of a branch target statement that appears in the same scoping unit as the *computed-goto-stmt*.

NOTE 8.20

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 $1 \leq i \leq n$ 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 ith 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.

8.2.3 Arithmetic IF statement

R847 *arithmetic-if-stmt* **is** IF (*scalar-numeric-expr*) *label* , *label* , *label*

C832 (R847) Each *label* shall be the label of a branch target statement that appears in the same scoping unit as the *arithmetic-if-stmt*.

C833 (R847) The *scalar-numeric-expr* shall not be of type complex.

NOTE 8.21

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 statement identified by the first label, the second label, or the third label is executed next depending on whether 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.

R848 *continue-stmt* **is** CONTINUE

8.4 STOP statement

R849 *stop-stmt* **is** STOP [*stop-code*]
R850 *stop-code* **is** *scalar-char-constant*
or *digit* [*digit* [*digit* [*digit* [*digit*]]]]

C834 (R850) *scalar-char-constant* shall be of type default character.

Execution of a STOP statement causes normal termination (2.3.4) of execution of the program. At the time of termination, the stop code, if any, is available in a processor-dependent manner. Leading zero digits in the stop code are not significant. If any exception (14) is signaling, the processor shall issue a warning indicating which exceptions are signaling; this warning shall be on the unit identified by the named constant ERROR_UNIT from the ISO_FORTRAN_ENV intrinsic module (13.8.2.2).

Section 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, FLUSH, WAIT, 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**.

A file is composed of either a sequence of file storage units (9.2.4) or a sequence of records, which provide an extra level of organization to the file. A file composed of records is called a **record file**. A file composed of file storage units is called a **stream file**. A processor may allow a file to be viewed both as a record file and as a stream file; in this case the relationship between the file storage units when viewed as a stream file and the records when viewed as a record file is processor dependent.

A file is either an external file (9.2) or an internal file (9.3).

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

NOTE 9.1

What is called a "record" in Fortran is commonly called a "logical record". There is no concept in Fortran of a "physical record."

9.1.1 Formatted record

A **formatted record** consists of a sequence of characters that are capable of representation in the processor; however, a processor may prohibit some control characters (3.1) from appearing in a formatted record. 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.

9.1.2 Unformatted record

An **unformatted record** consists of a sequence of values in a processor-dependent form and may contain data of any type or may contain no data. The length of an unformatted record is measured in file storage units (9.2.4) and depends on the output list (9.5.2) 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 shall be connected for sequential access. An endfile record is written implicitly to a file connected for sequential access when the most recent data transfer statement referring to the file is a data transfer output statement, no intervening file positioning statement referring to the file has been executed, and

(1) A REWIND or BACKSPACE statement references the unit to which the file is connected or

(2) The unit is closed, either explicitly by a CLOSE statement, implicitly by a program termination not caused by an error condition, or implicitly by another OPEN statement for the same unit.

An endfile record may occur only as the last record of a file. An endfile record does not have a length property.

NOTE 9.2

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 (9.7.1).

9.2 External files

An **external file** is any file that exists in a medium external to the program.

At any given time, there is a processor-dependent set of allowed **access methods**, a processor-dependent set of allowed **forms**, a processor-dependent set of allowed **actions**, and a processor-dependent set of allowed **record lengths** for a file.

NOTE 9.3

For example, the processor-dependent set of allowed actions for a printer would likely include the write action, but not the read action.

A file may have a name; a file that has a name is called a **named file**. The name of a named file is represented by a character string value. The set of allowable names for a file is processor dependent.

An external file that is connected to a unit has a **position** property (9.2.3).

NOTE 9.4

For more explanatory information on external files, see C.6.1.

9.2.1 File existence

At any given time, there is a processor-dependent set of external files that are said to **exist** for a program. A file may be known to the processor, yet not exist for a program at a particular time.

NOTE 9.5

> Security reasons may prevent a file from existing for a program. A newly created file may exist but contain no records.

To create a file means to cause a file to exist that did not exist previously. 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, FLUSH, or ENDFILE statement also may refer to a file that does not exist. Execution of a WRITE, PRINT, or ENDFILE statement referring to a preconnected file that does not exist creates the file.

9.2.2 File access

There are three methods of accessing the data of an external file: sequential, direct, and stream. Some files may have more than one allowed access method; other files may be restricted to one access method.

NOTE 9.6

> 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 a file is determined when the file is connected to a unit (9.4.3) or when the file is created if the file is preconnected (9.4.4).

9.2.2.1 Sequential access

Sequential access is a method of accessing the records of an external record file in order.

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 accessible by sequential access is the record whose record number is 1 for direct access. The second record accessible 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 shall not be read.

(2) The records of the file are either all formatted or all unformatted, except that the last record of the file may be an endfile record. Unless the previous reference to the file was a data transfer output statement, the last record, if any, of the file shall be an endfile record.

(3) The records of the file shall be read or written only by sequential access input/output statements.

9.2.2.2 Direct access

Direct access is a method of accessing the records of an external record file in arbitrary order.

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. The order of the records is the order of their record numbers.

NOTE 9.7

A record cannot be deleted; however, a record may be rewritten.

(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 shall not contain an endfile record.

(3) The records of the file shall be read or written 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, and if a READ statement for this connection is permitted.

(6) The records of the file shall not be read or written using list-directed formatting (10.9), namelist formatting (10.10), or a nonadvancing input/output statement (9.2.3.1).

9.2.2.3 Stream access

Stream access is a method of accessing the file storage units (9.2.4) of an external stream file.

The properties of an external file connected for stream access depend on whether the connection is for unformatted or formatted access.

When connected for unformatted stream access, an external file has the following properties:

(1) The file storage units of the file shall be read or written only by stream access input/output statements.

(2) Each file storage unit in the file is uniquely identified by a positive integer called the position. The first file storage unit in the file is at position 1. The position of each subsequent file storage unit is one greater than that of its preceding file storage unit.

(3) If it is possible to position the file, the file storage units need not be read or written in order of their position. For example, it might be permissible to write the file storage unit at position 3, even though the file storage units at positions 1 and 2 have not been written. Any file storage unit may be read from the file while it is connected to a unit, provided that the file storage unit has been written since the file was created, and if a READ statement for this connection is permitted.

When connected for formatted stream access, an external file has the following properties:

(1) Some file storage units of the file may contain record markers; this imposes a record structure on the file in addition to its stream structure. There might or might not be a record marker at the end of the file. If there is no record marker at the end of the file, the final record is incomplete.

(2) No maximum length (9.4.5.12) is applicable to these records.

(3) Writing an empty record with no record marker has no effect.

NOTE 9.8

Because the record structure is determined from the record markers that are stored in the file itself, an incomplete record at the end of the file is necessarily not empty.

(4) The file storage units of the file shall be read or written only by formatted stream access input/output statements.

(5) Each file storage unit in the file is uniquely identified by a positive integer called the position. The first file storage unit in the file is at position 1. The relationship between positions of successive file storage units is processor dependent; not all positive integers need correspond to valid positions.

(6) If it is possible to position the file, the file position can be set to a position that was previously identified by the POS= specifier in an INQUIRE statement.

NOTE 9.9

There may be some character positions in the file that do not correspond to characters written; this is because on some processors a record marker may be written to the file as a carriage-return/line-feed or other sequence. The means of determining the position in a file connected for stream access is via the POS= specifier in an INQUIRE statement (9.9.1.21).

(7) A processor may prohibit some control characters (3.1) from appearing in a formatted stream file.

9.2.3 File position

Execution of certain input/output statements affects the position of an external 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 or file storage unit. The **terminal point** is the position just after the last record or file storage unit. If there are no records or file storage units in the file, the initial point and the terminal point are the same position.

If a record 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.

For a file connected for stream access, the file position is either between two file storage units, at the initial point of the file, at the terminal point of the file, or undefined.

9.2.3.1 Advancing and nonadvancing input/output

An **advancing input/output statement** always positions a record file after the last record read or written, unless there is an error condition.

A **nonadvancing input/output statement** may position a record file at a character position within the current record, or a subsequent record (10.7.2). Using nonadvancing input/output, it is possible to read or write a record of the file by a sequence of input/output statements, each accessing a portion of the record. It is also possible to read variable-length records and be notified of their lengths. If a nonadvancing output statement leaves a file positioned within a current record and no further output statement is executed for the file before it is closed or a BACKSPACE, ENDFILE, or REWIND statement is executed for it, the effect is as if the output statement were the corresponding advancing output statement.

9.2.3.2 File position prior to data transfer

The positioning of the file prior to data transfer depends on the method of access: sequential, direct, or stream.

For sequential access on input, if there is a current record, the file position is not changed. Otherwise, the file is positioned at the beginning of the next record and this record becomes the current record. Input shall not occur if there is no next record or if there is a current record and the last data transfer statement accessing the file performed output.

If the file contains an endfile record, the file shall not be positioned after the endfile record prior to data transfer. However, a REWIND or BACKSPACE statement may be used to reposition the file.

For sequential access on output, if there is a current record, the file position is not changed and the current record becomes the last record of the file. Otherwise, a new record is created as the next record of the file; this new record becomes the last and current record of the file and the file is positioned at the beginning of this record.

For direct access, the file is positioned at the beginning of the record specified by the REC= specifier. This record becomes the current record.

For stream access, the file is positioned immediately before the file storage unit specified by the POS= specifier; if there is no POS= specifier, the file position is not changed.

File positioning for child data transfer statements is described in 9.5.3.7.

9.2.3.3 File position after data transfer

If an error condition (9.10) occurred, the position of the file is indeterminate. If no error condition occurred, but an end-of-file condition (9.10) occurred as a result of reading an endfile record, the file is positioned after the endfile record.

For unformatted stream access, if no error condition occurred, the file position is not changed. For unformatted stream output, if the file position exceeds the previous terminal point of the file, the terminal point is set to the file position.

NOTE 9.10

An unformatted stream output statement with a POS= specifier and an empty output list can have the effect of extending the terminal point of a file without actually writing any data.

For a formatted stream output statement, if no error condition occurred, the terminal point of the file is set to the highest-numbered position to which data was transferred by the statement.

NOTE 9.11

The highest-numbered position might not be the current one if the output involved T or TL edit descriptors (10.7.1.1).

For formatted stream input, if an end-of-file condition occurred, the file position is not changed.

For nonadvancing input, if no error condition or end-of-file condition occurred, but an end-of-record condition (9.10) occurred, the file is positioned after the record just read. If no error condition, end-of-file condition, or end-of-record condition occurred in a nonadvancing input statement, the file position is not changed. If no error condition occurred in a nonadvancing output statement, the file position is not changed.

In all other cases, the file is positioned after the record just read or written and that record becomes the preceding record.

9.2.4 File storage units

A **file storage unit** is the basic unit of storage in a stream file or an unformatted record file. It is the unit of file position for stream access, the unit of record length for unformatted files, and the unit of file size for all external files.

Every value in a stream file or an unformatted record file shall occupy an integer number of file storage units; if the stream or record file is unformatted, this number shall be the same for all scalar values of the same type and type parameters. The number of file storage units required for an item of a given type and type parameters may be determined using the IOLENGTH= specifier of the INQUIRE statement (9.9.3).

For a file connected for unformatted stream access, the processor shall not have alignment restrictions that prevent a value of any type from being stored at any positive integer file position.

The number of bits in a file storage unit is given by the constant FILE_STORAGE_SIZE (13.8.2.3) defined in the intrinsic module ISO_FORTRAN_ENV. It is recommended that the file storage unit be an 8-bit octet where this choice is practical.

NOTE 9.12

The requirement that every data value occupy an integer number of file storage units implies that data items inherently smaller than a file storage unit will require padding. This suggests that the file storage unit be small to avoid wasted space. Ideally, the file storage unit would be chosen such that padding is never required. A file storage unit of one bit would always meet this goal, but would likely be impractical because of the alignment requirements.

The prohibition on alignment restrictions prohibits the processor from requiring data alignments larger than the file storage unit.

The 8-bit octet is recommended as a good compromise that is small enough to accommodate the requirements of many applications, yet not so small that the data alignment requirements are likely to cause significant performance problems.

9.3 Internal files

Internal files provide a means of transferring and converting data from internal storage to internal storage.

An internal file is a record file with the following properties:

(1) The file is a variable of default, ASCII, or ISO 10646 character type that is not an array section with a vector subscript.

(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, 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 (6.2.2.2) or the array section (6.2.2.3). 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. The number of characters to be written shall not exceed the length of the record.

(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, except for child data transfer statements (9.5.3.7). This record becomes the current record.

(8) The initial value of a connection mode (9.4.1) is the value that would be implied by an initial OPEN statement without the corresponding keyword.

(9) Reading and writing records shall be accomplished only by sequential access formatted input/output statements.

(10) An internal file shall not be specified as the unit in a file connection statement, a file positioning statement, or a file inquiry statement.

9.4 File connection

A **unit**, specified by an *io-unit*, provides a means for referring to a file.

R901	*io-unit*	**is**	*file-unit-number*
		or	*
		or	*internal-file-variable*
R902	*file-unit-number*	**is**	*scalar-int-expr*
R903	*internal-file-variable*	**is**	*char-variable*

C901 (R903) The *char-variable* shall not be an array section with a vector subscript.

C902 (R903) The *char-variable* shall be of type default character, ASCII character, or ISO 10646 character.

A unit is either an external unit or an internal unit. An **external unit** is used to refer to an external file and is specified by an asterisk or a *file-unit-number* whose value is nonnegative or equal to one of the named constants INPUT_UNIT, OUTPUT_UNIT, or ERROR_UNIT of the ISO_FORTRAN_ENV module (13.8.2). An **internal unit** is used to refer to an internal file and is specified by an *internal-file-variable* or a *file-unit-number* whose value is equal to the unit argument of an active derived-type input/output procedure (9.5.3.7). The value of a *file-unit-number* shall identify a valid unit.

The external unit identified by a particular value of a *scalar-int-expr* is the same external unit in all program units of the program.

NOTE 9.13

In the example:

```
SUBROUTINE A
   READ (6) X
      ...
SUBROUTINE B
   N = 6
   REWIND N
```

NOTE 9.13 (cont.)

the value 6 used in both program units identifies the same external unit.

An asterisk identifies particular processor-dependent external units that are preconnected for formatted sequential access (9.5.3.2). These units are also identified by unit numbers defined by the named constants INPUT_UNIT and OUTPUT_UNIT of the ISO_FORTRAN_ENV module (13.8.2).

This standard identifies a processor-dependent external unit for the purpose of error reporting. This unit shall be preconnected for sequential formatted output. The processor may define this to be the same as the output unit identified by an asterisk. This unit is also identified by a unit number defined by the named constant ERROR_UNIT of the ISO_FORTRAN_ENV intrinsic module.

9.4.1 Connection modes

A connection for formatted input/output has several changeable modes: the blank interpretation mode (10.7.6), delimiter mode (10.9.2, 10.10.2.1), sign mode (10.7.4), decimal edit mode (10.7.8), I/O rounding mode (10.6.1.2.6), pad mode (9.5.3.4.2), and scale factor (10.7.5). A connection for unformatted input/output has no changeable modes.

Values for the modes of a connection are established when the connection is initiated. If the connection is initiated by an OPEN statement, the values are as specified, either explicitly or implicitly, by the OPEN statement. If the connection is initiated other than by an OPEN statement (that is, if the file is an internal file or preconnected file) the values established are those that would be implied by an initial OPEN statement without the corresponding keywords.

The scale factor cannot be explicitly specified in an OPEN statement; it is implicitly 0.

The modes of a connection to an external file may be changed by a subsequent OPEN statement that modifies the connection.

The modes of a connection may be temporarily changed by a corresponding keyword specifier in a data transfer statement or by an edit descriptor. Keyword specifiers take effect at the beginning of execution of the data transfer statement. Edit descriptors take effect when they are encountered in format processing. When a data transfer statement terminates, the values for the modes are reset to the values in effect immediately before the data transfer statement was executed.

9.4.2 Unit existence

At any given time, there is a processor-dependent set of external units that are said to exist for a program.

All input/output statements may refer to units that exist. The CLOSE, INQUIRE, and WAIT statements also may refer to units that do not exist.

9.4.3 Connection of a file to a unit

An external unit has a property of being **connected** or not connected. If connected, it refers to an external file. An external unit may become connected by preconnection or by the execution of an OPEN statement. The property of connection is symmetric; the unit is connected to a file if and only if the file is connected to the unit.

Every input/output statement except an OPEN, CLOSE, INQUIRE, or WAIT statement shall refer to 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 (9.2.1).

NOTE 9.14

An example is a preconnected external file that has not yet been written.

A unit shall not be connected to more than one file at the same time, and a file shall not be connected to more than one unit at the same time. However, means are provided to change the status of an external unit and to connect a unit to a different file.

This standard defines means of portable interoperation with C. C streams are described in 7.19.2 of the C standard. Whether a unit may be connected to a file that is also connected to a C stream is processor dependent. If the processor allows a unit to be connected to a file that is also connected to a C stream, the results of performing input/output operations on such a file are processor dependent. It is processor dependent whether the files connected to the units INPUT_UNIT, OUTPUT_UNIT, and ERROR_UNIT correspond to the predefined C text streams standard input, standard output, and standard error. If a procedure defined by means of Fortran and a procedure defined by means other than Fortran perform input/output operations on the same external file, the results are processor dependent. A procedure defined by means of Fortran and a procedure defined by means other than Fortran can perform input/output operations on different external files without interference.

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

NOTE 9.15

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 might be no means of reconnecting an unnamed file once it is disconnected.

An internal unit is always connected to the internal file designated by the variable that identifies the unit.

NOTE 9.16

For more explanatory information on file connection properties, see C.6.5.

9.4.4 Preconnection

Preconnection means that the unit is connected to a file at the beginning of execution of the program and therefore it may be specified in input/output statements without the prior execution of an OPEN statement.

9.4.5 The OPEN statement

An **OPEN statement** initiates or modifies the connection between an external file and a specified unit. 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 modes of a connection between a file and a unit.

An external unit may be connected by an OPEN statement in any program unit of a program and, once connected, a reference to it may appear in any program unit of the 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 such an OPEN statement, the file to be connected to the unit is the same as the file to which the unit is already 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 modes 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 connected, only the specifiers for changeable modes (9.4.1) may have values different from those currently in effect. If the POSITION= specifier appears in such an OPEN statement, the value specified shall not disagree with the current position of the file. If the STATUS= specifier is included in such an OPEN statement, it shall be specified with the value OLD. Execution of such an OPEN statement causes any new values of the specifiers for changeable modes to be in effect, but does not cause any change in any of the unspecified specifiers and the position of the file is unaffected. The ERR=, IOSTAT=, and IOMSG= specifiers from any previously executed OPEN statement have no effect on any currently executed OPEN statement.

A STATUS= specifier with a value of OLD is always allowed when the file to be connected to the unit is the same as the file to which the unit is connected. In this case, if the status of the file was SCRATCH before execution of the OPEN statement, the file will still be deleted when the unit is closed, and the file is still considered to have a status of SCRATCH.

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 =] *file-unit-number*
		or	ACCESS = *scalar-default-char-expr*
		or	ACTION = *scalar-default-char-expr*
		or	ASYNCHRONOUS = *scalar-default-char-expr*
		or	BLANK = *scalar-default-char-expr*
		or	DECIMAL = *scalar-default-char-expr*
		or	DELIM = *scalar-default-char-expr*
		or	ENCODING = *scalar-default-char-expr*
		or	ERR = *label*
		or	FILE = *file-name-expr*
		or	FORM = *scalar-default-char-expr*
		or	IOMSG = *iomsg-variable*
		or	IOSTAT = *scalar-int-variable*
		or	PAD = *scalar-default-char-expr*
		or	POSITION = *scalar-default-char-expr*
		or	RECL = *scalar-int-expr*
		or	ROUND = *scalar-default-char-expr*
		or	SIGN = *scalar-default-char-expr*
		or	STATUS = *scalar-default-char-expr*
R906	*file-name-expr*	**is**	*scalar-default-char-expr*
R907	*iomsg-variable*	**is**	*scalar-default-char-variable*

C903 (R905) No specifier shall appear more than once in a given *connect-spec-list*.

C904 (R905) A *file-unit-number* shall be specified; if the optional characters UNIT= are omitted, the *file-unit-number* shall be the first item in the *connect-spec-list*.

C905 (R905) The *label* used in the ERR= specifier shall be the statement label of a branch target statement that appears in the same scoping unit as the OPEN statement.

If the STATUS= specifier has the value NEW or REPLACE, the FILE= specifier shall appear. If the

STATUS= specifier has the value SCRATCH, the FILE= specifier shall not appear. If the STATUS= specifier has the value OLD, the FILE= specifier shall appear unless the unit is connected and the file connected to the unit exists.

A specifier that requires a *scalar-default-char-expr* may have a limited list of character values. These values are listed for each such specifier. Any trailing blanks are ignored. The value specified is without regard to case. Some specifiers have a default value if the specifier is omitted.

The IOSTAT=, ERR=, and IOMSG= specifiers are described in 9.10.

NOTE 9.17

An example of an OPEN statement is:

```
OPEN (10, FILE = 'employee.names', ACTION = 'READ', PAD = 'YES')
```

NOTE 9.18

For more explanatory information on the OPEN statement, see C.6.4.

9.4.5.1 ACCESS= specifier in the OPEN statement

The *scalar-default-char-expr* shall evaluate to SEQUENTIAL, DIRECT, or STREAM. The ACCESS= specifier specifies the access method for the connection of the file as being sequential, direct, or stream. If this specifier is omitted, the default value is SEQUENTIAL. For an existing file, the specified access method shall 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.

9.4.5.2 ACTION= specifier in the OPEN statement

The *scalar-default-char-expr* shall evaluate to READ, WRITE, or READWRITE. READ specifies that the WRITE, PRINT, and ENDFILE statements shall not refer to this connection. WRITE specifies that READ statements shall not refer to this connection. READWRITE permits any input/output statements to refer to this connection. If this specifier is omitted, the default value is processor dependent. If READWRITE is included in the set of allowable actions for a file, both READ and WRITE also shall be included in the set of allowed actions for that file. For an existing file, the specified action shall be included in the set of allowed actions for the file. For a new file, the processor creates the file with a set of allowed actions that includes the specified action.

9.4.5.3 ASYNCHRONOUS= specifier in the OPEN statement

The *scalar-default-char-expr* shall evaluate to YES or NO. If YES is specified, asynchronous input/output on the unit is allowed. If NO is specified, asynchronous input/output on the unit is not allowed. If this specifier is omitted, the default value is NO.

9.4.5.4 BLANK= specifier in the OPEN statement

The *scalar-default-char-expr* shall evaluate to NULL or ZERO. The BLANK= specifier is permitted only for a connection for formatted input/output. It specifies the current value of the blank interpretation mode (10.7.6, 9.5.1.5) for input for this connection. This mode has no effect on output. It is a changeable mode (9.4.1). If this specifier is omitted in an OPEN statement that initiates a connection, the default value is NULL.

9.4.5.5 DECIMAL= specifier in the OPEN statement

The *scalar-default-char-expr* shall evaluate to COMMA or POINT. The DECIMAL= specifier is permitted only for a connection for formatted input/output. It specifies the current value of the decimal edit mode (10.7.8, 9.5.1.6) for this connection. This is a changeable mode (9.4.1). If this specifier is omitted in an OPEN statement that initiates a connection, the default value is POINT.

9.4.5.6 DELIM= specifier in the OPEN statement

The *scalar-default-char-expr* shall evaluate to APOSTROPHE, QUOTE, or NONE. The DELIM= specifier is permitted only for a connection for formatted input/output. It specifies the current value of the delimiter mode (9.5.1.7) for list-directed (10.9.2) and namelist (10.10.2.1) output for the connection. This mode has no effect on input. It is a changeable mode (9.4.1). If this specifier is omitted in an OPEN statement that initiates a connection, the default value is NONE.

9.4.5.7 ENCODING= specifier in the OPEN statement

The *scalar-default-char-expr* shall evaluate to UTF-8 or DEFAULT. The ENCODING= specifier is permitted only for a connection for formatted input/output. The value UTF-8 specifies that the encoding form of the file is UTF-8 as specified by ISO/IEC 10646-1:2000. Such a file is called a **Unicode** file, and all characters therein are of ISO 10646 character type. The value UTF-8 shall not be specified if the processor does not support the ISO 10646 character type. The value DEFAULT specifies that the encoding form of the file is processor-dependent. If this specifier is omitted in an OPEN statement that initiates a connection, the default value is DEFAULT.

9.4.5.8 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. Any trailing blanks are ignored. The *file-name-expr* shall be a name that is allowed by the processor. If this specifier is omitted and the unit is not connected to a file, the STATUS= specifier shall be specified with a value of SCRATCH; in this case, the connection is made to a processor-dependent file. The interpretation of case is processor dependent.

9.4.5.9 FORM= specifier in the OPEN statement

The *scalar-default-char-expr* shall evaluate to FORMATTED or UNFORMATTED. The FORM= specifier determines whether the file is being connected for formatted or unformatted input/output. If this specifier is omitted, the default value is UNFORMATTED if the file is being connected for direct access or stream access, and the default value is FORMATTED if the file is being connected for sequential access. For an existing file, the specified form shall 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.

9.4.5.10 PAD= specifier in the OPEN statement

The *scalar-default-char-expr* shall evaluate to YES or NO. The PAD= specifier is permitted only for a connection for formatted input/output. It specifies the current value of the pad mode (9.5.3.4.2, 9.5.1.9) for input for this connection. This mode has no effect on output. It is a changeable mode (9.4.1). If this specifier is omitted in an OPEN statement that initiates a connection, the default value is YES.

NOTE 9.19

For nondefault character types, the blank padding character is processor dependent.

9.4.5.11 POSITION= specifier in the OPEN statement

The *scalar-default-char-expr* shall evaluate to ASIS, REWIND, or APPEND. The connection shall be for sequential or stream access. A new file is positioned at its initial point. REWIND positions an existing file at its initial point. APPEND positions an existing file such that the endfile record is the next record, if it has one. If an existing file does not have an endfile record, APPEND positions the file at its terminal point. ASIS leaves the position unchanged if the file exists and already is connected. ASIS leaves the position unspecified if the file exists but is not connected. If this specifier is omitted, the default value is ASIS.

9.4.5.12 RECL= specifier in the OPEN statement

The value of the RECL= specifier shall be positive. It specifies the length of each record in a file being connected for direct access, or specifies the maximum length of a record in a file being connected for sequential access. This specifier shall not appear when a file is being connected for stream access. This specifier shall appear when a file is being connected for direct access. If this specifier is omitted when a file is being connected for sequential access, the default value is processor dependent. If the file is being connected for formatted input/output, the length is the number of characters for all records that contain only characters of type default character. When a record contains any nondefault characters, the appropriate value for the RECL= specifier is processor dependent. If the file is being connected for unformatted input/output, the length is measured in file storage units. For an existing file, the value of the RECL= specifier shall 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.

9.4.5.13 ROUND= specifier in the OPEN statement

The *scalar-default-char-expr* shall evaluate to one of UP, DOWN, ZERO, NEAREST, COMPATIBLE, or PROCESSOR_DEFINED. The ROUND= specifier is permitted only for a connection for formatted input/output. It specifies the current value of the I/O rounding mode (10.6.1.2.6, 9.5.1.12) for this connection. This is a changeable mode (9.4.1). If this specifier is omitted in an OPEN statement that initiates a connection, the I/O rounding mode is processor dependent; it shall be one of the above modes.

NOTE 9.20

A processor is free to select any I/O rounding mode for the default mode. The mode might correspond to UP, DOWN, ZERO, NEAREST, or COMPATIBLE; or it might be a completely different I/O rounding mode.

9.4.5.14 SIGN= specifier in the OPEN statement

The *scalar-default-char-expr* shall evaluate to one of PLUS, SUPPRESS, or PROCESSOR_DEFINED. The SIGN= specifier is permitted only for a connection for formatted input/output. It specifies the current value of the sign mode (10.7.4, 9.5.1.13) for this connection. This is a changeable mode (9.4.1). If this specifier is omitted in an OPEN statement that initiates a connection, the default value is PROCESSOR_DEFINED.

9.4.5.15 STATUS= specifier in the OPEN statement

The *scalar-default-char-expr* shall evaluate to OLD, NEW, SCRATCH, REPLACE, or UNKNOWN. If OLD is specified, the file shall exist. If NEW is specified, the file shall not exist.

Successful execution of an OPEN statement with NEW specified creates the file and changes the status to OLD. If REPLACE is specified and the file does not already exist, the file is created and the status is changed to OLD. If REPLACE is specified and the file does exist, the file is deleted, a new file is created with the same name, and the status is changed to OLD. If SCRATCH is specified, the file is created and connected to the specified unit for use by the program but is deleted at the execution of a CLOSE statement referring to the same unit or at the normal termination of the program.

NOTE 9.21

SCRATCH shall not be specified with a named file.

If UNKNOWN is specified, the status is processor dependent. If this specifier is omitted, the default value is UNKNOWN.

9.4.6 The CLOSE statement

The **CLOSE statement** is used to terminate the connection of a specified unit to an external file.

Execution of a CLOSE statement for a unit may occur in any program unit of a 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 performs a wait operation for any pending asynchronous data transfer operations for the specified 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 program, either to the same file or to a different file. After a named file has been disconnected by execution of a CLOSE statement, it may be connected again within the same program, either to the same unit or to a different unit, provided that the file still exists.

At normal termination of execution of a program, 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 9.22

The effect is as though a CLOSE statement without a STATUS= specifier were executed on each connected unit.

R908	*close-stmt*	**is**	CLOSE (*close-spec-list*)
R909	*close-spec*	**is**	[UNIT =] *file-unit-number*
		or	IOSTAT = *scalar-int-variable*
		or	IOMSG = *iomsg-variable*
		or	ERR = *label*
		or	STATUS = *scalar-default-char-expr*

C906 (R909) No specifier shall appear more than once in a given *close-spec-list*.

C907 (R909) A *file-unit-number* shall be specified; if the optional characters UNIT= are omitted, the *file-unit-number* shall be the first item in the *close-spec-list*.

C908 (R909) The *label* used in the ERR= specifier shall be the statement label of a branch target statement that appears in the same scoping unit as the CLOSE statement.

The *scalar-default-char-expr* has a limited list of character values. Any trailing blanks are ignored. The value specified is without regard to case.

The IOSTAT=, ERR=, and IOMSG= specifiers are described in 9.10.

NOTE 9.23

An example of a CLOSE statement is:

```
CLOSE (10, STATUS = 'KEEP')
```

NOTE 9.24

For more explanatory information on the CLOSE statement, see C.6.6.

9.4.6.1 STATUS= specifier in the CLOSE statement

The *scalar-default-char-expr* shall evaluate to KEEP or DELETE. The STATUS= specifier determines the disposition of the file that is connected to the specified unit. KEEP shall 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 default value is KEEP, unless the file status prior to execution of the CLOSE statement is SCRATCH, in which case the default value is DELETE.

9.5 Data transfer statements

The **READ statement** is the data transfer input statement. The **WRITE statement** and the **PRINT statement** are the data transfer output statements.

R910	*read-stmt*	**is**	READ (*io-control-spec-list*) [*input-item-list*]
		or	READ *format* [, *input-item-list*]
R911	*write-stmt*	**is**	WRITE (*io-control-spec-list*) [*output-item-list*]
R912	*print-stmt*	**is**	PRINT *format* [, *output-item-list*]

NOTE 9.25

Examples of data transfer statements are:

```
   READ (6, *) SIZE
   READ 10, A, B
   WRITE (6, 10) A, S, J
   PRINT 10, A, S, J
10 FORMAT (2E16.3, I5)
```

9.5.1 Control information list

A **control information list** is an *io-control-spec-list*. It governs data transfer.

R913	*io-control-spec*	**is**	[UNIT =] *io-unit*
		or	[FMT =] *format*
		or	[NML =] *namelist-group-name*
		or	ADVANCE = *scalar-default-char-expr*
		or	ASYNCHRONOUS = *scalar-char-initialization-expr*
		or	BLANK = *scalar-default-char-expr*
		or	DECIMAL = *scalar-default-char-expr*
		or	DELIM = *scalar-default-char-expr*
		or	END = *label*
		or	EOR = *label*

or ERR = *label*
or ID = *scalar-int-variable*
or IOMSG = *iomsg-variable*
or IOSTAT = *scalar-int-variable*
or PAD = *scalar-default-char-expr*
or POS = *scalar-int-expr*
or REC = *scalar-int-expr*
or ROUND = *scalar-default-char-expr*
or SIGN = *scalar-default-char-expr*
or SIZE = *scalar-int-variable*

C909 (R913) No specifier shall appear more than once in a given *io-control-spec-list*.

C910 (R913) An *io-unit* shall be specified; if the optional characters UNIT= are omitted, the *io-unit* shall be the first item in the *io-control-spec-list*.

C911 (R913) A DELIM= or SIGN= specifier shall not appear in a *read-stmt*.

C912 (R913) A BLANK=, PAD=, END=, EOR=, or SIZE= specifier shall not appear in a *write-stmt*.

C913 (R913) The *label* in the ERR=, EOR=, or END= specifier shall be the statement label of a branch target statement that appears in the same scoping unit as the data transfer statement.

C914 (R913) A *namelist-group-name* shall be the name of a namelist group.

C915 (R913) A *namelist-group-name* shall not appear if an *input-item-list* or an *output-item-list* appears in the data transfer statement.

C916 (R913) An *io-control-spec-list* shall not contain both a *format* and a *namelist-group-name*.

C917 (R913) If *format* appears without a preceding FMT=, it shall be the second item in the *io-control-spec-list* and the first item shall be *io-unit*.

C918 (R913) If *namelist-group-name* appears without a preceding NML=, it shall be the second item in the *io-control-spec-list* and the first item shall be *io-unit*.

C919 (R913) If *io-unit* is not a *file-unit-number*, the *io-control-spec-list* shall not contain a REC= specifier or a POS= specifier.

C920 (R913) If the REC= specifier appears, an END= specifier shall not appear, a *namelist-group-name* shall not appear, and the *format*, if any, shall not be an asterisk.

C921 (R913) An ADVANCE= specifier may appear only in a formatted sequential or stream input/output statement with explicit format specification (10.1) whose control information list does not contain an *internal-file-variable* as the *io-unit*.

C922 (R913) If an EOR= specifier appears, an ADVANCE= specifier also shall appear.

C923 (R913) If a SIZE= specifier appears, an ADVANCE= specifier also shall appear.

C924 (R913) The *scalar-char-initialization-expr* in an ASYNCHRONOUS= specifier shall be of type default character and shall have the value YES or NO.

C925 (R913) An ASYNCHRONOUS= specifier with a value YES shall not appear unless *io-unit* is a *file-unit-number*.

C926 (R913) If an ID= specifier appears, an ASYNCHRONOUS= specifier with the value YES shall also appear.

C927 (R913) If a POS= specifier appears, the *io-control-spec-list* shall not contain a REC= specifier.

C928 (R913) If a DECIMAL=, BLANK=, PAD=, SIGN=, or ROUND= specifier appears, a *format* or *namelist-group-name* shall also appear.

C929 (R913) If a DELIM= specifier appears, either *format* shall be an asterisk or *namelist-group-name* shall appear.

A SIZE= specifier may appear only in an input statement that contains an ADVANCE= specifier with the value NO.

An EOR= specifier may appear only in an input statement that contains an ADVANCE= specifier with the value NO.

If the data transfer statement contains a *format* or *namelist-group-name*, the statement is a **formatted input/output statement**; otherwise, it is an **unformatted input/output statement**.

The ADVANCE=, ASYNCHRONOUS=, DECIMAL=, BLANK=, DELIM=, PAD=, SIGN=, and ROUND= specifiers have a limited list of character values. Any trailing blanks are ignored. The values specified are without regard to case.

The IOSTAT=, ERR=, EOR=, END=, and IOMSG= specifiers are described in 9.10.

NOTE 9.26

An example of a READ statement is:

```
READ (IOSTAT = IOS, UNIT = 6, FMT = '(10F8.2)') A, B
```

9.5.1.1 FMT= specifier in a data transfer statement

The FMT= specifier supplies a format specification or specifies list-directed formatting for a formatted input/output statement.

R914	*format*	**is**	*default-char-expr*
		or	*label*
		or	*

C930 (R914) The *label* shall be the label of a FORMAT statement that appears in the same scoping unit as the statement containing the FMT= specifier.

The *default-char-expr* shall evaluate to a valid format specification (10.1.1 and 10.1.2).

NOTE 9.27

A *default-char-expr* includes a character constant.

If *default-char-expr* is an array, it is treated as if all of the elements of the array were specified in array element order and were concatenated.

If *format* is *, the statement is a **list-directed input/output statement**.

NOTE 9.28

An example in which the format is a character expression is:

```
READ (6, FMT = "(" // CHAR_FMT // ")" )  X, Y, Z
```

NOTE 9.28 (cont.)

> where CHAR_FMT is a default character variable.

9.5.1.2 NML= specifier in a data transfer statement

The NML= specifier supplies the *namelist-group-name* (5.4). This name identifies a particular collection of data objects on which transfer is to be performed.

If a *namelist-group-name* appears, the statement is a **namelist input/output statement**.

9.5.1.3 ADVANCE= specifier in a data transfer statement

The *scalar-default-char-expr* shall evaluate to YES or NO. The ADVANCE= specifier determines whether advancing input/output occurs for this input/output statement. If YES is specified, advancing input/output occurs. If NO is specified, nonadvancing input/output occurs (9.2.3.1). If this specifier is omitted from an input/output statement that allows the specifier, the default value is YES.

9.5.1.4 ASYNCHRONOUS= specifier in a data transfer statement

The ASYNCHRONOUS= specifier determines whether this input/output statement is synchronous or asynchronous. If YES is specified, the statement and the input/output operation are said to be asynchronous. If NO is specified or if the specifier is omitted, the statement and the input/output operation are said to be synchronous.

Asynchronous input/output is permitted only for external files opened with an ASYNCHRONOUS= specifier with the value YES in the OPEN statement.

NOTE 9.29

> Both synchronous and asynchronous input/output are allowed for files opened with an ASYNCHRONOUS= specifier of YES. For other files, only synchronous input/output is allowed; this includes files opened with an ASYNCHRONOUS= specifier of NO, files opened without an ASYNCHRONOUS= specifier, preconnected files accessed without an OPEN statement, and internal files.
>
> The ASYNCHRONOUS= specifier value in a data transfer statement is an initialization expression because it effects compiler optimizations and, therefore, needs to be known at compile time.

The processor may perform an asynchronous data transfer operation asynchronously, but it is not required to do so. For each external file, records and file storage units read or written by asynchronous data transfer statements are read, written, and processed in the same order as they would have been if the data transfer statements were synchronous.

If a variable is used in an asynchronous data transfer statement as

(1) an item in an input/output list,
(2) a group object in a namelist, or
(3) a SIZE= specifier

the base object of the *data-ref* is implicitly given the ASYNCHRONOUS attribute in the scoping unit of the data transfer statement. This attribute may be confirmed by explicit declaration.

When an asynchronous input/output statement is executed, the set of storage units specified by the item list or NML= specifier, plus the storage units specified by the SIZE= specifier, is defined to be the pending input/output storage sequence for the data transfer operation.

NOTE 9.30

A pending input/output storage sequence is not necessarily a contiguous set of storage units.

A pending input/output storage sequence **affector** is a variable of which any part is associated with a storage unit in a pending input/output storage sequence.

9.5.1.5 BLANK= specifier in a data transfer statement

The *scalar-default-char-expr* shall evaluate to NULL or ZERO. The BLANK= specifier temporarily changes (9.4.1) the blank interpretation mode (10.7.6, 9.4.5.4) for the connection. If the specifier is omitted, the mode is not changed.

9.5.1.6 DECIMAL= specifier in a data transfer statement

The *scalar-default-char-expr* shall evaluate to COMMA or POINT. The DECIMAL= specifier temporarily changes (9.4.1) the decimal edit mode (10.7.8, 9.4.5.5) for the connection. If the specifier is omitted, the mode is not changed.

9.5.1.7 DELIM= specifier in a data transfer statement

The *scalar-default-char-expr* shall evaluate to APOSTROPHE, QUOTE, or NONE. The DELIM= specifier temporarily changes (9.4.1) the delimiter mode (10.9.2, 10.10.2.1, 9.4.5.6) for the connection. If the specifier is omitted, the mode is not changed.

9.5.1.8 ID= specifier in a data transfer statement

Successful execution of an asynchronous data transfer statement containing an ID= specifier causes the variable specified in the ID= specifier to become defined with a processor-dependent value. This value is referred to as the identifier of the data transfer operation. It can be used in a subsequent WAIT or INQUIRE statement to identify the particular data transfer operation.

If an error occurs during the execution of a data transfer statement containing an ID= specifier, the variable specified in the ID= specifier becomes undefined.

A child data transfer statement shall not specify the ID= specifier.

9.5.1.9 PAD= specifier in a data transfer statement

The *scalar-default-char-expr* shall evaluate to YES or NO. The PAD= specifier temporarily changes (9.4.1) the pad mode (9.5.3.4.2, 9.4.5.10) for the connection. If the specifier is omitted, the mode is not changed.

9.5.1.10 POS= specifier in a data transfer statement

The POS= specifier specifies the file position in file storage units. This specifier may appear in a data transfer statement only if the statement specifies a unit connected for stream access. A child data transfer statement shall not specify this specifier.

A processor may prohibit the use of POS= with particular files that do not have the properties necessary to support random positioning. A processor may also prohibit positioning a particular file to any position prior to its current file position if the file does not have the properties necessary to support such positioning.

NOTE 9.31

A file that represents connection to a device or data stream might not be positionable.

If the file is connected for formatted stream access, the file position specified by POS= shall be equal to either 1 (the beginning of the file) or a value previously returned by a POS= specifier in an INQUIRE statement for the file.

9.5.1.11 REC= specifier in a data transfer statement

The REC= specifier specifies the number of the record that is to be read or written. This specifier may appear only in an input/output statement that specifies a unit connected for direct access; it shall not appear in a child data transfer statement. If the control information list contains a REC= specifier, the statement is a **direct access input/output statement**. A child data transfer statement is a direct access data transfer statement if the parent is a direct access data transfer statement. Any other data transfer statement is a **sequential access input/output statement** or a **stream access input/output statement**, depending on whether the file connection is for sequential access or stream access.

9.5.1.12 ROUND= specifier in a data transfer statement

The *scalar-default-char-expr* shall evaluate to one of the values specified in 9.4.5.13. The ROUND= specifier temporarily changes (9.4.1) the I/O rounding mode (10.6.1.2.6, 9.4.5.13) for the connection. If the specifier is omitted, the mode is not changed.

9.5.1.13 SIGN= specifier in a data transfer statement

The *scalar-default-char-expr* shall evaluate to PLUS, SUPPRESS, or PROCESSOR_DEFINED. The SIGN= specifier temporarily changes (9.4.1) the sign mode (10.7.4, 9.4.5.14) for the connection. If the specifier is omitted, the mode is not changed.

9.5.1.14 SIZE= specifier in a data transfer statement

When a synchronous nonadvancing input statement terminates, the variable specified in the SIZE= specifier becomes defined with the count of the characters transferred by data edit descriptors during execution of the current input statement. Blanks inserted as padding (9.5.3.4.2) are not counted.

For asynchronous nonadvancing input, the storage units specified in the SIZE= specifier become defined with the count of the characters transferred when the corresponding wait operation is executed.

9.5.2 Data transfer input/output list

An input/output list specifies the entities whose values are transferred by a data transfer input/output statement.

R915	*input-item*	**is**	*variable*
		or	*io-implied-do*
R916	*output-item*	**is**	*expr*
		or	*io-implied-do*
R917	*io-implied-do*	**is**	(*io-implied-do-object-list* , *io-implied-do-control*)
R918	*io-implied-do-object*	**is**	*input-item*
		or	*output-item*
R919	*io-implied-do-control*	**is**	*do-variable* = *scalar-int-expr* , ■

■ *scalar-int-expr* [, *scalar-int-expr*]

C931 (R915) A variable that is an ***input-item*** shall not be a whole assumed-size array.

C932 (R915) A variable that is an ***input-item*** shall not be a procedure pointer.

C933 (R919) The ***do-variable*** shall be a named scalar variable of type integer.

C934 (R918) In an ***input-item-list***, an ***io-implied-do-object*** shall be an ***input-item***. In an ***output-item-list***, an ***io-implied-do-object*** shall be an ***output-item***.

C935 (R916) An expression that is an ***output-item*** shall not have a value that is a procedure pointer.

An ***input-item*** shall not appear as, nor be associated with, the ***do-variable*** of any ***io-implied-do*** that contains the ***input-item***.

NOTE 9.32

A constant, an expression involving operators or function references, or an expression enclosed in parentheses may appear as an output list item but shall not appear as an input list item.

If an input item is a pointer, it shall be associated with a definable target and data are transferred from the file to the associated target. If an output item is a pointer, it shall be associated with a target and data are transferred from the target to the file.

NOTE 9.33

Data transfers always involve the movement of values between a file and internal storage. A pointer as such cannot be read or written. Therefore, a pointer shall not appear as an item in an input/output list unless it is associated with a target that can receive a value (input) or can deliver a value (output).

If an input item or an output item is allocatable, it shall be allocated.

A list item shall not be polymorphic unless it is processed by a user-defined derived-type input/output procedure (9.5.3.7).

The ***do-variable*** of an ***io-implied-do*** that is in another ***io-implied-do*** shall not appear as, nor be associated with, the ***do-variable*** of the containing ***io-implied-do***.

The following rules describing whether to expand an input/output list item are re-applied to each expanded list item until none of the rules apply.

If an array appears as an input/output list item, it is treated as if the elements, if any, were specified in array element order (6.2.2.2). However, no element of that array may affect the value of any expression in the ***input-item***, nor may any element appear more than once in an ***input-item***.

NOTE 9.34

For example:

```
INTEGER A (100), J (100)
   ...
READ *, A (A)                              ! Not allowed
READ *, A (LBOUND (A, 1) : UBOUND (A, 1))  ! Allowed
READ *, A (J)                              ! Allowed if no two elements
                                           !    of J have the same value
A(1) = 1; A(10) = 10
```

NOTE 9.34 (cont.)

```
READ *, A (A (1) : A (10))              ! Not allowed
```

If a list item of derived type in an unformatted input/output statement is not processed by a user-defined derived-type input/output procedure (9.5.3.7), and if any subobject of that list item would be processed by a user-defined derived-type input/output procedure, the list item is treated as if all of the components of the object were specified in the list in component order (4.5.3.5); those components shall be accessible in the scoping unit containing the input/output statement and shall not be pointers or allocatable.

An effective input/output list item of derived type in an unformatted input/output statement is treated as a single value in a processor-dependent form unless the list item or a subobject thereof is processed by a user-defined derived-type input/output procedure (9.5.3.7).

NOTE 9.35

The appearance of a derived-type object as an input/output list item in an unformatted input/output statement is not equivalent to the list of its components.

Unformatted input/output involving derived-type list items forms the single exception to the rule that the appearance of an aggregate list item (such as an array) is equivalent to the appearance of its expanded list of component parts. This exception permits the processor greater latitude in improving efficiency or in matching the processor-dependent sequence of values for a derived-type object to similar sequences for aggregate objects used by means other than Fortran. However, formatted input/output of all list items and unformatted input/output of list items other than those of derived types adhere to the above rule.

If a list item of derived type in a formatted input/output statement is not processed by a user-defined derived-type input/output procedure, that list item is treated as if all of the components of the list item were specified in the list in component order; those components shall be accessible in the scoping unit containing the input/output statement and shall not be pointers or allocatable.

If a derived-type list item is not treated as a list of its individual components, that list item's ultimate components shall not have the POINTER or ALLOCATABLE attribute unless that list item is processed by a user-defined derived-type input/output procedure.

The scalar objects resulting when a data transfer statement's list items are expanded according to the rules in this section for handling array and derived-type list items are called **effective items**. Zero-sized arrays and *io-implied-do*s with an iteration count of zero do not contribute to the effective list items. A scalar character item of zero length is an effective list item.

NOTE 9.36

In a formatted input/output statement, edit descriptors are associated with effective list items, which are always scalar. The rules in 9.5.2 determine the set of effective list items corresponding to each actual list item in the statement. These rules may have to be applied repetitively until all of the effective list items are scalar items.

For an *io-implied-do*, the loop initialization and execution is the same as for a DO construct (8.1.6.4).

An input/output list shall not contain an item of nondefault character type if the input/output statement specifies an internal file of default character type. An input/output list shall not contain an item of nondefault character type other than ISO 10646 or ASCII character type if the input/output statement specifies an internal file of ISO 10646 character type. An input/output list shall not contain a character item of any character type other than ASCII character type if the input/output statement specifies an internal file of ASCII character type.

NOTE 9.37

An example of an output list with an implied DO is:

```
WRITE (LP, FMT = '(10F8.2)') (LOG (A (I)), I = 1, N + 9, K), G
```

9.5.3 Execution of a data transfer input/output statement

Execution of a WRITE or PRINT statement for a file that does not exist creates the file unless an error condition occurs.

The effect of executing a synchronous data transfer input/output statement shall be as if the following operations were performed in the order specified:

(1) Determine the direction of data transfer.
(2) Identify the unit.
(3) Perform a wait operation for all pending input/output operations for the unit. If an error, end-of-file, or end-of-record condition occurs during any of the wait operations, steps 4 through 8 are skipped for the current data transfer statement.
(4) Establish the format if one is specified.
(5) Position the file prior to data transfer (9.2.3.2) unless the statement is a child data transfer statement (9.5.3.7).
(6) Transfer data between the file and the entities specified by the input/output list (if any) or namelist.
(7) Determine whether an error, end-of-file, or end-of-record condition has occurred.
(8) Position the file after data transfer (9.2.3.3) unless the statement is a child data transfer statement (9.5.3.7).
(9) Cause any variable specified in a SIZE= specifier to become defined.
(10) If an error, end-of-file, or end-of-record condition occurred, processing continues as specified in 9.10; otherwise any variable specified in an IOSTAT= specifier is assigned the value zero.

The effect of executing an asynchronous data transfer input/output statement shall 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 (9.2.3.2).
(5) Establish the set of storage units identified by the input/output list. For a READ statement, this might require some or all of the data in the file to be read if an input variable is used as a *scalar-int-expr* in an *io-implied-do-control* in the input/output list, as a *subscript*, *substring-range*, *stride*, or is otherwise referenced.
(6) Initiate an asynchronous data transfer between the file and the entities specified by the input/output [illegible]

may occur during the execution of this data transfer statement.

(8) Position the file as if the data transfer had finished (9.2.3.3).

(9) Cause any variable specified in a SIZE= specifier to become undefined.

(10) If an error, end-of-file, or end-of-record condition occurred, processing continues as specified in 9.10; otherwise any variable specified in an IOSTAT= specifier is assigned the value zero.

For an asynchronous data transfer statement, the data transfers may occur during execution of the statement, during execution of the corresponding wait operation, or anywhere between. The data transfer operation is considered to be pending until a corresponding wait operation is performed.

For asynchronous output, a pending input/output storage sequence affector (9.5.1.4) shall not be redefined, become undefined, or have its pointer association status changed.

For asynchronous input, a pending input/output storage sequence affector shall not be referenced, become defined, become undefined, become associated with a dummy argument that has the VALUE attribute, or have its pointer association status changed.

Error, end-of-file, and end-of-record conditions in an asynchronous data transfer operation may occur during execution of either the data transfer statement or the corresponding wait operation. If an ID= specifier does not appear in the initiating data transfer statement, the conditions may occur during the execution of any subsequent data transfer or wait operation for the same unit. When a condition occurs for a previously executed asynchronous data transfer statement, a wait operation is performed for all pending data transfer operations on that unit. When a condition occurs during a subsequent statement, any actions specified by IOSTAT=, IOMSG=, ERR=, END=, and EOR= specifiers for that statement are taken.

NOTE 9.38

Because end-of-file and error conditions for asynchronous data transfer statements without an ID= specifier may be reported by the processor during the execution of a subsequent data transfer statement, it may be impossible for the user to determine which input/output statement caused the condition. Reliably detecting which READ statement caused an end-of-file condition requires that all asynchronous READ statements for the unit include an ID= specifier.

9.5.3.1 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 any, or specified within the file itself for namelist input. 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, or by the *namelist-group-name* for namelist output.

9.5.3.2 Identifying a unit

A data transfer input/output statement that contains an input/output control list includes a UNIT= specifier that identifies an external or internal unit. A READ statement that does not contain an input/output control list specifies a particular processor-dependent 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-dependent unit, which is the same as the unit identified by * in a WRITE statement. Thus, each data transfer input/output statement identifies an external or internal unit.

The unit identified by a data transfer input/output statement shall be connected to a file when execution of the statement begins.

NOTE 9.39

The file may be preconnected.

9.5.3.3 Establishing a format

If the input/output control list contains * as a format, list-directed formatting is established. If *namelist-group-name* appears, namelist formatting is established. If no *format* or *namelist-group-name* is specified, unformatted data transfer is established. Otherwise, the format specification identified by the FMT= specifier is established.

On output, if an internal file has been specified, a format specification that is in the file or is associated with the file shall not be specified.

9.5.3.4 Data transfer

Data are transferred between the file and the entities specified by the input/output list or namelist. The list items are processed in the order of the input/output list for all data transfer input/output statements except namelist formatted data transfer statements. The list items for a namelist input statement are processed in the order of the entities specified within the input records. The list items for a namelist output statement are processed in the order in which the variables are specified in the *namelist-group-object-list*. Effective items are derived from the input/output list items as described in 9.5.2.

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.

NOTE 9.40

In the example,

```
READ (N) N, X (N)
```

the old value of N identifies the unit, but the new value of N is the subscript of X.

All values following the *name*= part of the namelist entity (10.10) within the input records are transmitted to the matching entity specified in the *namelist-group-object-list* prior to processing any succeeding entity within the input record for namelist input statements. If an entity is specified more than once within the input record during a namelist 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, shall not contain any portion of an established format specification.

If the input/output item is a pointer, data are transferred between the file and the associated target.

If an internal file has been specified, an input/output list item shall not be in the file or associated with the file.

NOTE 9.41

The file is a data object.

A DO variable becomes defined and its iteration count established 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 shall be defined.

9.5.3.4.1 Unformatted data transfer

During unformatted data transfer, data are transferred without editing between the file and the entities specified by the input/output list. If the file is connected for sequential or direct access, exactly one record is read or written.

Objects of intrinsic or derived types may be transferred by means of an unformatted data transfer statement.

A value in the file is stored in a contiguous sequence of file storage units, beginning with the file storage unit immediately following the current file position.

After each value is transferred, the current file position is moved to a point immediately after the last file storage unit of the value.

On input from a file connected for sequential or direct access, the number of file storage units required by the input list shall be less than or equal to the number of file storage units in the record.

On input, if the file storage units transferred do not contain a value with the same type and type parameters as the input list entity, then the resulting value of the entity is processor-dependent except in the following cases:

(1) A complex list entity may correspond to two real values of the same kind stored in the file, or vice-versa.

(2) A default character list entity of length n may correspond to n default characters stored in the file, regardless of the length parameters of the entities that were written to these storage units of the file. If the file is connected for stream input, the characters may have been written by formatted stream output.

On output to a file connected for unformatted direct access, the output list shall not specify more values than can fit into the record. 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 unformatted sequential access, the record is created with a length sufficient to hold the values from the output list. This length shall be one of the set of allowed record lengths for the file and shall not exceed the value specified in the RECL= specifier, if any, of the OPEN statement that established the connection.

If the file is not connected for unformatted input/output, unformatted data transfer is prohibited.

The unit specified shall be an external unit.

9.5.3.4.2 Formatted data transfer

During formatted data transfer, data are transferred with editing between the file and the entities specified by the input/output list or by the *namelist-group-name*. Format control is initiated and editing is performed as described in Section 10.

The current record and possibly additional records are read or written.

Values may be transferred by means of a formatted data transfer statement to or from objects of intrinsic or derived types. In the latter case, the transfer is in the form of values of intrinsic types to or from the components of intrinsic types that ultimately comprise these structured objects unless the derived-type list item is processed by a user-defined derived-type input/output procedure (9.5.3.7).

If the file is not connected for formatted input/output, formatted data transfer is prohibited.

During advancing input when the pad mode has the value NO, the input list and format specification shall not require more characters from the record than the record contains.

During advancing input when the pad mode has the value YES, blank characters are supplied by the

A data transfer statement that includes a derived-type list item and that causes a user-defined derived-type input/output procedure to be invoked is called a **parent data transfer statement**. A data transfer statement that is executed while a parent data transfer statement is being processed and that specifies the unit passed into a user-defined derived-type input/output procedure is called a **child data transfer statement**.

NOTE 9.42

A user-defined derived-type input/output procedure will usually contain child data transfer statements that read values from or write values to the current record or at the current file position. The effect of executing the user-defined derived-type input/output procedure is similar to that of substituting the list items from any child data transfer statements into the parent data transfer statement's list items, along with similar substitutions in the format specification.

NOTE 9.43

A particular execution of a READ, WRITE or PRINT statement can be both a parent and a child data transfer statement. A user-defined derived-type input/output procedure can indirectly call itself or another user-defined derived-type input/output procedure by executing a child data transfer statement containing a list item of derived type, where a matching interface is accessible for that derived type. If a user-defined derived-type input/output procedure calls itself indirectly in this manner, it shall be declared RECURSIVE.

A child data transfer statement is processed differently from a nonchild data transfer statement in the following ways:

- Executing a child data transfer statement does not position the file prior to data transfer.
- An unformatted child data transfer statement does not position the file after data transfer is complete.

9.5.3.7.2 User-defined derived-type input/output procedures

For a particular derived type and a particular set of kind type parameter values, there are four possible sets of characteristics for user-defined derived-type input/output procedures; one each for formatted input, formatted output, unformatted input, and unformatted output. The user need not supply all four procedures. The procedures are specified to be used for derived-type input/output by interface blocks (12.3.2.1) or by generic bindings (4.5.4), with a *dtio-generic-spec* (R1208).

In the four interfaces, which specify the characteristics of user-defined procedures for derived-type input/output, the following syntax term is used:

R920 *dtv-type-spec* **is** TYPE(*derived-type-spec*)
or CLASS(*derived-type-spec*)

C936 (R920) If *derived-type-spec* specifies an extensible type, the CLASS keyword shall be used; otherwise, the TYPE keyword shall be used.

C937 (R920) All length type parameters of *derived-type-spec* shall be assumed.

If the *dtio-generic-spec* is READ (FORMATTED), the characteristics shall be the same as those specified by the following interface:

```
SUBROUTINE my_read_routine_formatted                &
                               (dtv,                &
```

```
                                    unit,                  &
                                    iotype, v_list,        &
                                    iostat, iomsg)
            ! the derived-type value/variable
            dtv-type-spec, INTENT(INOUT) :: dtv
```

```
                                        iostat, iomsg)
            ! the derived-type value/variable
            dtv-type-spec, INTENT(IN) :: dtv
            INTEGER, INTENT(IN) :: unit
            INTEGER, INTENT(OUT) :: iostat
            CHARACTER (LEN=*), INTENT(INOUT) :: iomsg
          END
```

The actual specific procedure names (the `my_..._routine_...` procedure names above) are not significant. In the discussion here and elsewhere, the dummy arguments in these interfaces are referred by the names given above; the names are, however, arbitrary.

In addition to the characteristics specified by the above interfaces, the `dtv` dummy argument may optionally have the VOLATILE attribute.

When a user-defined derived-type input/output procedure is invoked, the processor shall pass a `unit` argument that has a value as follows:

- If the parent data transfer statement uses a *file-unit-number*, the value of the `unit` argument shall be that of the *file-unit-number*.
- If the parent data transfer statement is a WRITE statement with an asterisk unit or a PRINT statement, the `unit` argument shall have the same value as the OUTPUT_UNIT named constant of the ISO_FORTRAN_ENV intrinsic module (13.8.2).
- If the parent data transfer statement is a READ statement with an asterisk unit or a READ statement without an *io-control-spec-list*, the `unit` argument shall have the same value as the INPUT_UNIT named constant of the ISO_FORTRAN_ENV intrinsic module (13.8.2).
- Otherwise the parent data transfer statement must access an internal file, in which case the `unit` argument shall have a processor-dependent negative value.

NOTE 9.44

Because the unit argument value will be negative when the parent data transfer statement specifies an internal file, a user-defined derived-type input/output procedure should not execute an INQUIRE statement without checking that the `unit` argument is nonnegative or is equal to one of the named constants INPUT_UNIT, OUTPUT_UNIT, or ERROR_UNIT of the ISO_FORTRAN_ENV intrinsic module (13.8.2).

For formatted data transfer, the processor shall pass an `iotype` argument that has a value as follows:

- "LISTDIRECTED" if the parent data transfer statement specified list directed formatting,
- "NAMELIST" if the parent data transfer statement specified namelist formatting, or
- "DT" concatenated with the *char-literal-constant*, if any, of the edit descriptor, if the parent data transfer statement contained a format specification and the list item's corresponding edit descriptor was a DT edit descriptor.

If the parent data transfer statement is a READ statement, the `dtv` dummy argument is argument associated with the effective list item that caused the user-defined derived-type input procedure to be invoked, as if the effective list item were an actual argument in this procedure reference (2.5.6).

If the parent data transfer statement is a WRITE or PRINT statement, the processor shall provide the value of the effective list item in the `dtv` dummy argument.

If the *v-list*

necessarily at the beginning of a record.

A record positioning edit descriptor, such as TL and TR, used on `unit` by a child data transfer statement shall not cause the record position to be positioned before the record position at the time the user-defined derived-type input/output procedure was invoked.

NOTE 9.47

A robust user-defined derived-type input/output procedure may wish to use INQUIRE to determine the settings of BLANK=, PAD=, ROUND=, DECIMAL=, and DELIM= for an external unit. The INQUIRE provides values as specified in 9.9.

Neither a parent nor child data transfer statement shall be asynchronous.

A user-defined derived-type input/output procedure, and any procedures invoked therefrom, shall not define, nor cause to become undefined, any storage location referenced by any input/output list item, the corresponding format, or any specifier in any active parent data transfer statement, except through the `dtv` argument.

NOTE 9.48

A child data transfer statement shall not specify the ID=, POS=, or REC= specifiers in an input/output control list.

NOTE 9.49

A simple example of derived type formatted output follows. The derived type variable `chairman` has two components. The type and an associated write formatted procedure are defined in a module so as to be accessible from wherever they might be needed. It would also be possible to check that `iotype` indeed has the value 'DT' and to set `iostat` and `iomsg` accordingly.

```
MODULE p

  TYPE :: person
    CHARACTER (LEN=20) :: name
    INTEGER :: age
  CONTAINS
    GENERIC :: WRITE(FORMATTED) => pwf
  END TYPE person

CONTAINS

  SUBROUTINE pwf (dtv,unit,iotype,vlist,iostat,iomsg)
! argument definitions
    CLASS(person), INTENT(IN) :: dtv
    INTEGER, INTENT(IN) :: unit
    CHARACTER (LEN=*), INTENT(IN) :: iotype
    INTEGER, INTENT(IN) :: vlist(:)
    INTEGER, INTENT(OUT) :: iostat
    CHARACTER (LEN=*), INTENT(INOUT) :: iomsg
! local variable
    CHARACTER (LEN=9) :: pfmt

!   vlist(1) and (2) are to be used as the field widths of the two
!   components of the derived type variable.  First set up the format to
!   be used for output.
```

NOTE 9.49 (cont.)

```
    WRITE(pfmt,'(A,I2,A,I2,A)' ) '(A',  vlist(1), ',I',  vlist(2),  ')'

!   now the basic output statement
    WRITE(unit, FMT=pfmt, IOSTAT=iostat) dtv%name, dtv%age

  END SUBROUTINE pwf

END MODULE p

PROGRAM
  USE p
  INTEGER id, members
  TYPE (person) :: chairman
  ...
  WRITE(6, FMT="(I2, DT (15,6), I5)" ) id, chairman, members
! this writes a record with four fields, with lengths 2, 15, 6, 5
! respectively

END PROGRAM
```

NOTE 9.50

In the following example, the variables of the derived type `node` form a linked list, with a single value at each node. The subroutine `pwf` is used to write the values in the list, one per line.

```
MODULE p

  TYPE node
    INTEGER :: value = 0
    TYPE (NODE), POINTER :: next_node => NULL ( )
  CONTAINS
    GENERIC :: WRITE(FORMATTED) => pwf
  END TYPE node

CONTAINS

  RECURSIVE SUBROUTINE pwf (dtv,unit,iotype,vlist,iostat,iomsg)
! Write the chain of values, each on a separate line in I9 format.
    CLASS(node), INTENT(IN) :: dtv
    INTEGER, INTENT(IN) :: unit
    CHARACTER (LEN=*), INTENT(IN) :: iotype
    INTEGER, INTENT(IN) :: vlist(:)
    INTEGER, INTENT(OUT) :: iostat
    CHARACTER (LEN=*), INTENT(INOUT) :: iomsg

    WRITE(unit,'(i9 /)', IOSTAT = iostat) dtv%value
    IF(iostat/=0) RETURN
    IF(ASSOCIATED(dtv%next_node)) WRITE(unit,'(dt)', IOSTAT=iostat) dtv%next_node
  END SUBROUTINE pwf

END MODULE p
```

9.5.3.7.3 Resolving derived-type input/output procedure references

A suitable generic interface for user-defined derived-type input/output of an effective item is one that has a *dtio-generic-spec* that is appropriate to the direction (read or write) and form (formatted or unformatted) of the data transfer as specified in 9.5.3.7, and has a specific interface whose `dtv` argument is compatible with the effective item according to the rules for argument association in 12.4.1.2.

When an effective item (9.5.2) that is of derived-type is encountered during a data transfer, user-defined derived-type input/output occurs if both of the following conditions are true:

(1) The circumstances of the input/output are such that user-defined derived-type input/output is permitted; that is, either

- (a) the transfer was initiated by a list-directed, namelist, or unformatted input/output statement, or
- (b) a format specification is supplied for the input/output statement, and the edit descriptor corresponding to the effective item is a DT edit descriptor.

(2) A suitable user-defined derived-type input/output procedure is available; that is, either

- (a) the declared type of the effective item has a suitable generic type-bound procedure, or
- (b) a suitable generic interface is accessible.

If (2a) is true, the procedure referenced is determined as for explicit type-bound procedure references (12.4); that is, the binding with the appropriate specific interface is located in the declared type of the effective item, and the corresponding binding in the dynamic type of the effective item is selected.

If (2a) is false and (2b) is true, the reference is to the procedure identified by the appropriate specific interface in the interface block. This reference shall not be to a dummy procedure that is not present, or to a disassociated procedure pointer.

9.5.4 Termination of data transfer statements

Termination of an input/output data transfer statement occurs when any of the following conditions are met:

(1) Format processing encounters a data edit descriptor and there are no remaining elements in the *input-item-list* or *output-item-list*.
(2) Unformatted or list-directed data transfer exhausts the *input-item-list* or *output-item-list*.
(3) Namelist output exhausts the *namelist-group-object-list*.
(4) An error condition occurs.
(5) An end-of-file condition occurs.
(6) A slash (/) is encountered as a value separator (10.9, 10.10) in the record being read during list-directed or namelist input.
(7) An end-of-record condition occurs during execution of a nonadvancing input statement (9.10).

9.6 Waiting on pending data transfer

Execution of an asynchronous data transfer statement in which neither an error, end-of-record, nor end-of-file condition occurs initiates a pending data transfer operation. There may be multiple pending data transfer operations for the same or multiple units simultaneously. A pending data transfer operation remains pending until a corresponding wait operation is performed. A wait operation may be performed by a WAIT, INQUIRE, CLOSE, or file positioning statement.

9.6.1 WAIT statement

A WAIT statement performs a wait operation for specified pending asynchronous data transfer operations.

NOTE 9.51

The CLOSE, INQUIRE, and file positioning statements may also perform wait operations.

R921	*wait-stmt*	**is**	WAIT (*wait-spec-list*)
R922	*wait-spec*	**is**	[UNIT =] *file-unit-number*
		or	END = *label*
		or	EOR = *label*
		or	ERR = *label*
		or	ID = *scalar-int-variable*
		or	IOMSG = *iomsg-variable*
		or	IOSTAT = *scalar-int-variable*

C938 (R922) No specifier shall appear more than once in a given *wait-spec-list*.

C939 (R922) A *file-unit-number* shall be specified; if the optional characters UNIT= are omitted, the *file-unit-number* shall be the first item in the *wait-spec-list*.

C940 (R922) The *label* in the ERR=, EOR=, or END= specifier shall be the statement label of a branch target statement that appears in the same scoping unit as the WAIT statement.

The IOSTAT=, ERR=, EOR=, END=, and IOMSG= specifiers are described in 9.10.

The value of the variable specified in the ID= specifier shall be the identifier of a pending data transfer operation for the specified unit. If the ID= specifier appears, a wait operation for the specified data transfer operation is performed. If the ID= specifier is omitted, wait operations for all pending data transfers for the specified unit are performed.

Execution of a WAIT statement specifying a unit that does not exist, has no file connected to it, or was not opened for asynchronous input/output is permitted, provided that the WAIT statement has no ID= specifier; such a WAIT statement does not cause an error or end-of-file condition to occur.

NOTE 9.52

An EOR= specifier has no effect if the pending data transfer operation is not a nonadvancing read. And END= specifier has no effect if the pending data transfer operation is not a READ.

9.6.2 Wait operation

A wait operation completes the processing of a pending data transfer operation. Each wait operation completes only a single data transfer operation, although a single statement may perform multiple wait operations.

If the actual data transfer is not yet complete, the wait operation first waits for its completion. If the data transfer operation is an input operation that completed without error, the storage units of the input/output storage sequence then become defined with the values as described in 9.5.1.14 and 9.5.3.4.

If any error, end-of-file, or end-of-record conditions occur, the applicable actions specified by the IOSTAT=, IOMSG=, ERR=, END=, and EOR= specifiers of the statement that performs the wait operation are taken.

If an error or end-of-file condition occurs during a wait operation for a unit, the processor performs a wait operation for all pending data transfer operations for that unit.

NOTE 9.53

Error, end-of-file, and end-of-record conditions may be raised either during the data transfer statement that initiates asynchronous input/output, a subsequent asynchronous data transfer statement for the same unit, or during the wait operation. If such conditions are raised during a data transfer statement, they trigger actions according to the IOSTAT=, ERR=, END=, and EOR= specifiers of that statement; if they are raised during the wait operation, the actions are in accordance with the specifiers of the statement that performs the wait operation.

After completion of the wait operation, the data transfer operation and its input/output storage sequence are no longer considered to be pending.

9.7 File positioning statements

R923	*backspace-stmt*	**is**	BACKSPACE *file-unit-number*
		or	BACKSPACE (*position-spec-list*)
R924	*endfile-stmt*	**is**	ENDFILE *file-unit-number*
		or	ENDFILE (*position-spec-list*)
R925	*rewind-stmt*	**is**	REWIND *file-unit-number*
		or	REWIND (*position-spec-list*)

A file that is connected for direct access shall not be referred to by a BACKSPACE, ENDFILE, or REWIND statement. A file that is connected for unformatted stream access shall not be referred to by a BACKSPACE statement. A file that is connected with an ACTION= specifier having the value READ shall not be referred to by an ENDFILE statement.

R926	*position-spec*	**is**	[UNIT =] *file-unit-number*
		or	IOMSG = *iomsg-variable*
		or	IOSTAT = *scalar-int-variable*
		or	ERR = *label*

C941 (R926) No specifier shall appear more than once in a given *position-spec-list*.

C942 (R926) A *file-unit-number* shall be specified; if the optional characters UNIT= are omitted, the *file-unit-number* shall be the first item in the *position-spec-list*.

C943 (R926) The *label* in the ERR= specifier shall be the statement label of a branch target statement that appears in the same scoping unit as the file positioning statement.

The IOSTAT=, ERR=, and IOMSG= specifiers are described in 9.10.

Execution of a file positioning statement performs a wait operation for all pending asynchronous data transfer operations for the specified unit.

9.7.1 BACKSPACE statement

Execution of a BACKSPACE statement causes the file connected to the specified unit to be positioned before the current record if there is a current record, or before the preceding record if there is no current record. If the file is at its initial point, the position of the file is not changed.

NOTE 9.54

If the preceding record is an endfile record, the file is positioned before the endfile record.

If a BACKSPACE statement causes the implicit writing of an endfile record, the file is positioned before the record that precedes the endfile record.

Backspacing a file that is connected but does not exist is prohibited.

Backspacing over records written using list-directed or namelist formatting is prohibited.

NOTE 9.55

An example of a BACKSPACE statement is:

```
BACKSPACE (10, IOSTAT = N)
```

9.7.2 ENDFILE statement

Execution of an ENDFILE statement for a file connected for sequential access writes an endfile record as the next record of the file. The file is then positioned after the endfile record, which becomes the last record of the file. If the file also may 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 for a file connected for sequential access, a BACKSPACE or REWIND statement shall be used to reposition the file prior to execution of any data transfer input/output statement or ENDFILE statement.

Execution of an ENDFILE statement for a file connected for stream access causes the terminal point of the file to become equal to the current file position. Only file storage units before the current position are considered to have been written; thus only those file storage units may be subsequently read. Subsequent stream output statements may be used to write further data to the file.

Execution of an ENDFILE statement for a file that is connected but does not exist creates the file; if the file is connected for sequential access, it is created prior to writing the endfile record.

NOTE 9.56

An example of an ENDFILE statement is:

```
ENDFILE K
```

9.7.3 REWIND statement

Execution of a REWIND statement causes the specified file to be positioned at its initial point.

NOTE 9.57

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 and has no effect on any file.

NOTE 9.58

An example of a REWIND statement is:

```
REWIND 10
```

9.8 FLUSH statement

The form of the FLUSH statement is:

R927	*flush-stmt*	**is**	FLUSH *file-unit-number*
		or	FLUSH (*flush-spec-list*)
R928	*flush-spec*	**is**	[UNIT =] *file-unit-number*
		or	IOSTAT = *scalar-int-variable*
		or	IOMSG = *iomsg-variable*
		or	ERR = *label*

C944 (R928) No specifier shall appear more than once in a given *flush-spec-list*.

C945 (R928) A *file-unit-number* shall be specified; if the optional characters UNIT= are omitted from the unit specifier, the *file-unit-number* shall be the first item in the *flush-spec-list*.

C946 (R928) The *label* in the ERR= specifier shall be the statement label of a branch target statement that appears in the same scoping unit as the flush statement.

The IOSTAT=, IOMSG= and ERR= specifiers are described in 9.10. The IOSTAT= variable shall be set to a processor-dependent positive value if an error occurs, to zero if the processor-dependent flush operation was successful, or to a processor-dependent negative value if the flush operation is not supported for the unit specified.

Execution of a FLUSH statement causes data written to an external file to be available to other processes, or causes data placed in an external file by means other than Fortran to be available to a READ statement. The action is processor dependent.

Execution of a FLUSH statement for a file that is connected but does not exist is permitted and has no effect on any file. A FLUSH statement has no effect on file position.

NOTE 9.59

Because this standard does not specify the mechanism of file storage, the exact meaning of the flush operation is not precisely defined. The intention is that the flush operation should make all data written to a file available to other processes or devices, or make data recently added to a file by other processes or devices available to the program via a subsequent read operation. This is commonly called "flushing I/O buffers".

NOTE 9.60

An example of a FLUSH statement is:

```
FLUSH( 10, IOSTAT=N)
```

9.9 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 three forms of the INQUIRE statement: **inquire by file**, which uses the FILE= specifier, **inquire by unit**, which uses the UNIT= specifier, and **inquire by output list**, which uses only the IOLENGTH= specifier. All specifier 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.

R929	*inquire-stmt*	**is**	INQUIRE (*inquire-spec-list*)
		or	INQUIRE (IOLENGTH = *scalar-int-variable*) ■
			■ *output-item-list*

NOTE 9.61

Examples of INQUIRE statements are:

```
INQUIRE (IOLENGTH = IOL) A (1:N)
INQUIRE (UNIT = JOAN, OPENED = LOG_01, NAMED = LOG_02, &
   FORM = CHAR_VAR, IOSTAT = IOS)
```

9.9.1 Inquiry specifiers

Unless constrained, the following inquiry specifiers may be used in either of the inquire by file or inquire by unit forms of the INQUIRE statement:

R930	*inquire-spec*	**is**	[UNIT =] *file-unit-number*
		or	FILE = *file-name-expr*
		or	ACCESS = *scalar-default-char-variable*
		or	ACTION = *scalar-default-char-variable*
		or	ASYNCHRONOUS = *scalar-default-char-variable*
		or	BLANK = *scalar-default-char-variable*
		or	DECIMAL = *scalar-default-char-variable*
		or	DELIM = *scalar-default-char-variable*
		or	DIRECT = *scalar-default-char-variable*
		or	ENCODING = *scalar-default-char-variable*
		or	ERR = *label*
		or	EXIST = *scalar-default-logical-variable*
		or	FORM = *scalar-default-char-variable*
		or	FORMATTED = *scalar-default-char-variable*
		or	ID = *scalar-int-variable*
		or	IOMSG = *iomsg-variable*
		or	IOSTAT = *scalar-int-variable*
		or	NAME = *scalar-default-char-variable*
		or	NAMED = *scalar-default-logical-variable*
		or	NEXTREC = *scalar-int-variable*
		or	NUMBER = *scalar-int-variable*
		or	OPENED = *scalar-default-logical-variable*
		or	PAD = *scalar-default-char-variable*
		or	PENDING = *scalar-default-logical-variable*
		or	POS = *scalar-int-variable*
		or	POSITION = *scalar-default-char-variable*
		or	READ = *scalar-default-char-variable*
		or	READWRITE = *scalar-default-char-variable*
		or	RECL = *scalar-int-variable*
		or	ROUND = *scalar-default-char-variable*
		or	SEQUENTIAL = *scalar-default-char-variable*
		or	SIGN = *scalar-default-char-variable*
		or	SIZE = *scalar-int-variable*
		or	STREAM = *scalar-default-char-variable*
		or	UNFORMATTED = *scalar-default-char-variable*

or WRITE = *scalar-default-char-variable*

C947 (R930) No specifier shall appear more than once in a given *inquire-spec-list*.

C948 (R930) An *inquire-spec-list* shall contain one FILE= specifier or one UNIT= specifier, but not both.

C949 (R930) In the inquire by unit form of the INQUIRE statement, if the optional characters UNIT= are omitted, the *file-unit-number* shall be the first item in the *inquire-spec-list*.

C950 (R930) If an ID= specifier appears, a PENDING= specifier shall also appear.

The value of *file-unit-number* shall be nonnegative or equal to one of the named constants INPUT_UNIT, OUTPUT_UNIT, or ERROR_UNIT of the ISO_FORTRAN_ENV intrinsic module (13.8.2).

When a returned value of a specifier other than the NAME= specifier is of type character, the value returned is in upper case.

If an error condition occurs during execution of an INQUIRE statement, all of the inquiry specifier variables become undefined, except for variables in the IOSTAT= and IOMSG= specifiers (if any).

The IOSTAT=, ERR=, and IOMSG= specifiers are described in 9.10.

9.9.1.1 FILE= specifier in the INQUIRE statement

The value of the *file-name-expr* in the FILE= specifier specifies the name of the file being inquired about. The named file need not exist or be connected to a unit. The value of the *file-name-expr* shall be of a form acceptable to the processor as a file name. Any trailing blanks are ignored. The interpretation of case is processor dependent.

9.9.1.2 ACCESS= specifier in the INQUIRE statement

The *scalar-default-char-variable* in the ACCESS= specifier is assigned the value SEQUENTIAL if the file is connected for sequential access, DIRECT if the file is connected for direct access, or STREAM if the file is connected for stream access. If there is no connection, it is assigned the value UNDEFINED.

9.9.1.3 ACTION= specifier in the INQUIRE statement

The *scalar-default-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 READWRITE if it is connected for both input and output. If there is no connection, the *scalar-default-char-variable* is assigned the value UNDEFINED.

9.9.1.4 ASYNCHRONOUS= specifier in the INQUIRE statement

The *scalar-default-char-variable* in the ASYNCHRONOUS= specifier is assigned the value YES if the file is connected and asynchronous input/output on the unit is allowed; it is assigned the value NO if the file is connected and asynchronous input/output on the unit is not allowed. If there is no connection, the *scalar-default-char-variable* is assigned the value UNDEFINED.

9.9.1.5 BLANK= specifier in the INQUIRE statement

The *scalar-default-char-variable* in the BLANK= specifier is assigned the value ZERO or NULL, corresponding to the blank interpretation mode in effect for a connection for formatted input/output. If there is no connection, or if the connection is not for formatted input/output, the *scalar-default-char-variable* is assigned the value UNDEFINED.

9.9.1.6 DECIMAL= specifier in the INQUIRE statement

The *scalar-default-char-variable* in the DECIMAL= specifier is assigned the value COMMA or POINT, corresponding to the decimal edit mode in effect for a connection for formatted input/output. If there is no connection, or if the connection is not for formatted input/output, the *scalar-default-char-variable* is assigned the value UNDEFINED.

9.9.1.7 DELIM= specifier in the INQUIRE statement

The *scalar-default-char-variable* in the DELIM= specifier is assigned the value APOSTROPHE, QUOTE, or NONE, corresponding to the delimiter mode in effect for a connection for formatted input/output. If there is no connection or if the connection is not for formatted input/output, the *scalar-default-char-variable* is assigned the value UNDEFINED.

9.9.1.8 DIRECT= specifier in the INQUIRE statement

The *scalar-default-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.

9.9.1.9 ENCODING= specifier in the INQUIRE statement

The *scalar-default-char-variable* in the ENCODING= specifier is assigned the value UTF-8 if the file is connected for formatted input/output with an encoding form of UTF-8, and is assigned the value UNDEFINED if the file is connected for unformatted input/output. If there is no connection, it is assigned the value UTF-8 if the processor is able to determine that the encoding form of the file is UTF-8. If the processor is unable to determine the encoding form of the file, the variable is assigned the value UNKNOWN.

NOTE 9.62

The value assigned may be something other than UTF-8, UNDEFINED, or UNKNOWN if the processor supports other specific encoding forms (e.g. UTF-16BE).

9.9.1.10 EXIST= specifier in the INQUIRE statement

Execution of an INQUIRE by file statement causes the *scalar-default-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 by unit statement causes true to be assigned if the specified unit exists; otherwise, false is assigned.

9.9.1.11 FORM= specifier in the INQUIRE statement

The *scalar-default-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.

9.9.1.12 FORMATTED= specifier in the INQUIRE statement

The *scalar-default-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.

9.9.1.13 ID= specifier in the INQUIRE statement

The value of the variable specified in the ID= specifier shall be the identifier of a pending data transfer operation for the specified unit. This specifier interacts with the PENDING= specifier (9.9.1.20).

9.9.1.14 NAME= specifier in the INQUIRE statement

The *scalar-default-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 9.63

> 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. However, the value returned shall be suitable for use as the value of the *file-name-expr* in the FILE= specifier in an OPEN statement.
>
> The processor may return a file name qualified by a user identification, device, directory, or other relevant information.

The case of the characters assigned to *scalar-default-char-variable* is processor dependent.

9.9.1.15 NAMED= specifier in the INQUIRE statement

The *scalar-default-logical-variable* in the NAMED= specifier is assigned the value true if the file has a name; otherwise, it is assigned the value false.

9.9.1.16 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 from or written to the file connected for direct access. If the file is connected but no records have been read or written since the connection, the *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, the *scalar-int-variable* becomes undefined. If there are pending data transfer operations for the specified unit, the value assigned is computed as if all the pending data transfers had already completed.

9.9.1.17 NUMBER= specifier in the INQUIRE statement

The *scalar-int-variable* in the NUMBER= specifier is assigned the value of the external unit number of the unit that is connected to the file. If there is no unit connected to the file, the value -1 is assigned.

9.9.1.18 OPENED= specifier in the INQUIRE statement

Execution of an INQUIRE by file statement causes the *scalar-default-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 by unit statement causes the *scalar-default-logical-variable* to be assigned the value true if the specified unit is connected to a file; otherwise, false is assigned.

9.9.1.19 PAD= specifier in the INQUIRE statement

The *scalar-default-char-variable* in the PAD= specifier is assigned the value YES or NO, corresponding to the pad mode in effect for a connection for formatted input/output. If there is no connection or if the connection is not for formatted input/output, the *scalar-default-char-variable* is assigned the value UNDEFINED.

9.9.1.20 PENDING= specifier in the INQUIRE statement

The PENDING= specifier is used to determine whether or not previously pending asynchronous data transfers are complete. A data transfer operation is previously pending if it is pending at the beginning of execution of the INQUIRE statement.

If an ID= specifier appears and the specified data transfer operation is complete, then the variable specified in the PENDING= specifier is assigned the value false and the INQUIRE statement performs the wait operation for the specified data transfer.

If the ID= specifier is omitted and all previously pending data transfer operations for the specified unit are complete, then the variable specified in the PENDING= specifier is assigned the value false and the INQUIRE statement performs wait operations for all previously pending data transfers for the specified unit.

In all other cases, the variable specified in the PENDING= specifier is assigned the value true and no wait operations are performed; in this case the previously pending data transfers remain pending after the execution of the INQUIRE statement.

NOTE 9.64

The processor has considerable flexibility in defining when it considers a transfer to be complete. Any of the following approaches could be used:

(1) The INQUIRE statement could consider an asynchronous data transfer to be incomplete until after the corresponding wait operation. In this case PENDING= would always return true unless there were no previously pending data transfers for the unit.

(2) The INQUIRE statement could wait for all specified data transfers to complete and then always return false for PENDING=.

(3) The INQUIRE statement could actually test the state of the specified data transfer operations.

9.9.1.21 POS= specifier in the INQUIRE statement

The *scalar-int-variable* in the POS= specifier is assigned the number of the file storage unit immediately following the current position of a file connected for stream access. If the file is positioned at its terminal position, the variable is assigned a value one greater than the number of the highest-numbered file storage unit in the file. If the file is not connected for stream access or if the position of the file is indeterminate because of previous error conditions, the variable becomes undefined.

9.9.1.22 POSITION= specifier in the INQUIRE statement

The *scalar-default-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 before its endfile record or at its terminal point, and ASIS if the file is connected without changing its position. If there is no connection or if the file is connected for direct access, the *scalar-default-char-variable* is assigned the value UNDEFINED. If the file has been repositioned since the connection, the *scalar-default-char-variable* is assigned a processor-dependent value, which shall not be REWIND unless the file is positioned at its initial point and shall not be APPEND unless the file is positioned so that its endfile record is the next record or at its terminal point if it has no endfile record.

9.9.1.23 READ= specifier in the INQUIRE statement

The *scalar-default-char-variable* in the READ= specifier is assigned the value YES if READ is included in the set of allowed actions for the file, NO if READ is not included in the set of allowed actions for the file, and UNKNOWN if the processor is unable to determine whether or not READ is included in the set of allowed actions for the file.

9.9.1.24 READWRITE= specifier in the INQUIRE statement

The *scalar-default-char-variable* in the READWRITE= specifier is assigned the value YES if READWRITE is included in the set of allowed actions for the file, NO if READWRITE is not included in the set of allowed actions for the file, and UNKNOWN if the processor is unable to determine whether or not READWRITE is included in the set of allowed actions for the file.

9.9.1.25 RECL= specifier in the INQUIRE statement

The *scalar-int-variable* in the RECL= specifier is assigned the value of the record length of a file connected for direct access, or the value of the maximum record length for a file connected for sequential access. If the file is connected for formatted input/output, the length is the number of characters for all records that contain only characters of type default character. If the file is connected for unformatted input/output, the length is measured in file storage units. If there is no connection, or if the connection is for stream access, the *scalar-int-variable* becomes undefined.

9.9.1.26 ROUND= specifier in the INQUIRE statement

The *scalar-default-char-variable* in the ROUND= specifier is assigned the value UP, DOWN, ZERO, NEAREST, COMPATIBLE, or PROCESSOR_DEFINED, corresponding to the I/O rounding mode in effect for a connection for formatted input/output. If there is no connection or if the connection is not for formatted input/output, the *scalar-default-char-variable* is assigned the value UNDEFINED. The processor shall return the value PROCESSOR_DEFINED only if the I/O rounding mode currently in effect behaves differently than the UP, DOWN, ZERO, NEAREST, and COMPATIBLE modes.

9.9.1.27 SEQUENTIAL= specifier in the INQUIRE statement

The *scalar-default-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.

9.9.1.28 SIGN= specifier in the INQUIRE statement

The *scalar-default-char-variable* in the SIGN= specifier is assigned the value PLUS, SUPPRESS, or PROCESSOR_DEFINED, corresponding to the sign mode in effect for a connection for formatted input/output. If there is no connection, or if the connection is not for formatted input/output, the *scalar-default-char-variable* is assigned the value UNDEFINED.

9.9.1.29 SIZE= specifier in the INQUIRE statement

The *scalar-int-variable* in the SIZE= specifier is assigned the size of the file in file storage units. If the file size cannot be determined, the variable is assigned the value -1.

For a file that may be connected for stream access, the file size is the number of the highest-numbered file storage unit in the file.

For a file that may be connected for sequential or direct access, the file size may be different from the number of storage units implied by the data in the records; the exact relationship is processor-dependent.

9.9.1.30 STREAM= specifier in the INQUIRE statement

The *scalar-default-char-variable* in the STREAM= specifier is assigned the value YES if STREAM is included in the set of allowed access methods for the file, NO if STREAM 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 STREAM is included in the set of allowed access methods for the file.

9.9.1.31 UNFORMATTED= specifier in the INQUIRE statement

The *scalar-default-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.

9.9.1.32 WRITE= specifier in the INQUIRE statement

The *scalar-default-char-variable* in the WRITE= specifier is assigned the value YES if WRITE is included in the set of allowed actions for the file, NO if WRITE is not included in the set of allowed actions for the file, and UNKNOWN if the processor is unable to determine whether or not WRITE is included in the set of allowed actions for the file.

9.9.2 Restrictions on inquiry specifiers

The *inquire-spec-list* in an INQUIRE by file statement shall contain exactly one FILE= specifier and shall not contain a UNIT= specifier. The *inquire-spec-list* in an INQUIRE by unit statement shall contain exactly one UNIT= specifier and shall 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.

9.9.3 Inquire by output list

The inquire by output list form of the INQUIRE statement has only an IOLENGTH= specifier and an output list.

The *scalar-int-variable* in the IOLENGTH= specifier is assigned the processor-dependent number of file storage units that would be required to store the data of the output list in an unformatted file. The value shall 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.

The output list in an INQUIRE statement shall not contain any derived-type list items that require a user-defined derived-type input/output procedure as described in section 9.5.2. If a derived-type list item appears in the output list, the value returned for the IOLENGTH= specifier assumes that no user-defined derived-type input/output procedure will be invoked.

9.10 Error, end-of-record, and end-of-file conditions

The set of input/output error conditions is processor dependent.

An **end-of-record condition** occurs when a nonadvancing input statement attempts to transfer data from a position beyond the end of the current record, unless the file is a stream file and the current record is at the end of the file (an end-of-file condition occurs instead).

An **end-of-file condition** occurs in the following cases:

(1) When an endfile record is encountered during the reading of a file connected for sequential access.

(2) When an attempt is made to read a record beyond the end of an internal file.
(3) When an attempt is made to read beyond the end of a stream file.

An end-of-file condition may occur at the beginning of execution of an input statement. An end-of-file condition also may occur during execution of a formatted input statement when more than one record is required by the interaction of the input list and the format. An end-of-file condition also may occur during execution of a stream input statement.

9.10.1 Error conditions and the ERR= specifier

If an error condition occurs during execution of an input/output statement, the position of the file becomes indeterminate.

If an error condition occurs during execution of an input/output statement that contains neither an ERR= nor IOSTAT= specifier, execution of the program is terminated. If an error condition occurs during execution of an input/output statement that contains either an ERR= specifier or an IOSTAT= specifier then

(1) Processing of the input/output list, if any, terminates,
(2) If the statement is a data transfer statement or the error occurs during a wait operation, all *do-variable*s in the statement that initiated the transfer become undefined,
(3) If an IOSTAT= specifier appears, the *scalar-int-variable* in the IOSTAT= specifier becomes defined as specified in 9.10.4,
(4) If an IOMSG= specifier appears, the *iomsg-variable* becomes defined as specified in 9.10.5,
(5) If the statement is a READ statement and it contains a SIZE= specifier, the *scalar-int-variable* in the SIZE= specifier becomes defined as specified in 9.5.1.14,
(6) If the statement is a READ statement or the error condition occurs in a wait operation for a transfer initiated by a READ statement, all input items or namelist group objects in the statement that initiated the transfer become undefined, and
(7) If an ERR= specifier appears, execution continues with the statement labeled by the *label* in the ERR= specifier.

9.10.2 End-of-file condition and the END= specifier

If an end-of-file condition occurs during execution of an input/output statement that contains neither an END= specifier nor an IOSTAT= specifier, execution of the program is terminated. If an end-of-file condition occurs during execution of an input/output statement that contains either an END= specifier or an IOSTAT= specifier, and an error condition does not occur then

(1) Processing of the input list, if any, terminates,
(2) If the statement is a data transfer statement or the error occurs during a wait operation, all *do-variable*s in the statement that initiated the transfer become undefined,
(3) If the statement is a READ statement or the end-of-file condition occurs in a wait operation for a transfer initiated by a READ statement, all input list items or namelist group objects in the statement that initiated the transfer become undefined,
(4) If the file specified in the input statement is an external record file, it is positioned after the endfile record,
(5) If an IOSTAT= specifier appears, the *scalar-int-variable* in the IOSTAT= specifier becomes defined as specified in 9.10.4,
(6) If an IOMSG= specifier appears, the *iomsg-variable* becomes defined as specified in 9.10.5, and
(7) If an END= specifier appears, execution continues with the statement labeled by the *label* in the END= specifier.

9.10.3 End-of-record condition and the EOR= specifier

If an end-of-record condition occurs during execution of an input/output statement that contains neither an EOR= specifier nor an IOSTAT= specifier, execution of the program is terminated. If an end-of-record condition occurs during execution of an input/output statement that contains either an EOR= specifier or an IOSTAT= specifier, and an error condition does not occur then

(1) If the pad mode has the value YES, the record is padded with blanks to satisfy the input list item (9.5.3.4.2) and corresponding data edit descriptor that requires more characters than the record contains. If the pad mode has the value NO, the input list item becomes undefined.

(2) Processing of the input list, if any, terminates,

(3) If the statement is a data transfer statement or the error occurs during a wait operation, all *do-variable*s in the statement that initiated the transfer become undefined,

(4) The file specified in the input statement is positioned after the current record,

(5) If an IOSTAT= specifier appears, the *scalar-int-variable* in the IOSTAT= specifier becomes defined as specified in 9.10.4,

(6) If an IOMSG= specifier appears, the *iomsg-variable* becomes defined as specified in 9.10.5,

(7) If a SIZE= specifier appears, the *scalar-int-variable* in the SIZE= specifier becomes defined as specified in (9.5.1.14), and

(8) If an EOR= specifier appears, execution continues with the statement labeled by the *label* in the EOR= specifier.

9.10.4 IOSTAT= specifier

Execution of an input/output statement containing the IOSTAT= specifier causes the *scalar-int-variable* in the IOSTAT= specifier to become defined with

(1) A zero value if neither an error condition, an end-of-file condition, nor an end-of-record condition occurs,

(2) A processor-dependent positive integer value if an error condition occurs,

(3) The processor-dependent negative integer value of the constant IOSTAT_END (13.8.2.5) if an end-of-file condition occurs and no error condition occurs, or

(4) The processor-dependent negative integer value of the constant IOSTAT_EOR (13.8.2.6) if an end-of-record condition occurs and no error condition or end-of-file condition occurs.

NOTE 9.65

An end-of-file condition may occur only for sequential or stream input and an end-of-record condition may occur only for nonadvancing input.

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
```

9.10.5 IOMSG= specifier

If an error, end-of-file, or end-of-record condition occurs during execution of an input/output statement, the processor shall assign an explanatory message to *iomsg-variable*. If no such condition occurs, the processor shall not change the value of *iomsg-variable*.

9.11 Restrictions 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 shall not refer to the unit.

An input/output statement that is executed while another input/output statement is being executed is called a **recursive input/output statement**.

A recursive input/output statement shall not identify an external unit except that a child data transfer statement may identify its parent data transfer statement external unit.

An input/output statement shall not cause the value of any established format specification to be modified.

A recursive input/output statement shall not modify the value of any internal unit except that a recursive WRITE statement may modify the internal unit identified by that recursive WRITE statement.

The value of a specifier in an input/output statement shall not depend on any *input-item*, *io-implied-do do-variable*, or on the definition or evaluation of any other specifier in the *io-control-spec-list* or *inquire-spec-list* in that statement.

The value of any subscript or substring bound of a variable that appears in a specifier in an input/output statement shall not depend on any *input-item*, *io-implied-do do-variable*, or on the definition or evaluation of any other specifier in the *io-control-spec-list* or *inquire-spec-list* in that statement.

In a data transfer statement, the variable specified in an IOSTAT=, IOMSG=, or SIZE= specifier, if any, shall not be associated with any entity in the data transfer input/output list (9.5.2) or *namelist-group-object-list*, nor with a *do-variable* of an *io-implied-do* in the data transfer input/output list.

In a data transfer statement, if a variable specified in an IOSTAT=, IOMSG=, or SIZE= specifier is an array element reference, its subscript values shall not be affected by the data transfer, the *io-implied-do* processing, or the definition or evaluation of any other specifier in the *io-control-spec-list*.

A variable that may become defined or undefined as a result of its use in a specifier in an INQUIRE statement, or any associated entity, shall not appear in another specifier in the same INQUIRE statement.

A STOP statement shall not be executed during execution of an input/output statement.

NOTE 9.66

Restrictions on the evaluation of expressions (7.1.8) prohibit certain side effects.

Section 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 characters of a sequence of formatted records.

A FMT= specifier (9.5.1.1) 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 FMT= specifier alternatively may be an asterisk (*), which indicates list-directed formatting (10.9). Namelist formatting (10.10) may be indicated by specifying a *namelist-group-name* instead of a *format*.

10.1 Explicit format specification methods

Explicit format specification may be given

(1) In a FORMAT statement or

(2) In a character expression.

10.1.1 FORMAT statement

R1001	*format-stmt*	**is**	FORMAT *format-specification*
R1002	*format-specification*	**is**	([*format-item-list*])

C1001 (R1001) The *format-stmt* shall be labeled.

C1002 (R1002) The comma used to separate *format-items* in a *format-item-list* may be omitted

(1) Between a P edit descriptor and an immediately following F, E, EN, ES, D, or G edit descriptor (10.7.5), possibly preceded by a repeat specifier,

(2) Before a slash edit descriptor when the optional repeat specification is not present (10.7.2),

(3) After a slash edit descriptor, or

(4) Before or after a colon edit descriptor (10.7.3)

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 interpretation of the format specification, except within a character string edit descriptor (10.8).

NOTE 10.1

Examples of FORMAT statements are:

```
5     FORMAT (1PE12.4, I10)
9     FORMAT (I12, /, ' Dates: ', 2 (2I3, I5))
```

10.1.2 Character format specification

A character expression used as a *format* in a formatted input/output statement shall evaluate to a character string whose leading part is a valid format specification.

NOTE 10.2

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 shall be defined at the time the input/output statement is executed, and shall 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 interpretation of the format specification.

If the *format* is a character array, it is treated as if all of the elements of the array were specified in array element order and were concatenated. However, if a *format* is a character array element, the format specification shall be entirely within that array element.

NOTE 10.3

If a character constant is used as a *format* in an input/output statement, care shall be taken that the value of the character constant is a valid format specification. In particular, if a format specification delimited by apostrophes contains a character constant edit descriptor delimited with apostrophes, two apostrophes shall be written to delimit the edit descriptor and four apostrophes shall be written for each apostrophe that occurs within the 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, 1X, 'ISN''T', 1X, I1)
    WRITE (6, '(1X, I1, 1X, ''ISN''''T'', 1X, I1)') 2, 3
    WRITE (6, '(A)') ' 2 ISN''T 3'
```

Doubling of internal apostrophes usually can be avoided by using quotation marks to delimit the format specification and doubling of internal quotation marks usually can be avoided by using apostrophes as delimiters.

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-literal-constant*

C1003 (R1004) *r* shall be positive.

C1004 (R1004) *r* shall not have a kind parameter specified for it.

The integer literal constant *r* is called a **repeat specification.**

10.2.1 Edit descriptors

An **edit descriptor** is a **data edit descriptor**, a **control edit descriptor**, or a **character string edit descriptor**.

R1005	*data-edit-desc*	**is**	I *w* [. *m*]
		or	B *w* [. *m*]
		or	O *w* [. *m*]
		or	Z *w* [. *m*]
		or	F *w* . *d*
		or	E *w* . *d* [E *e*]
		or	EN *w* . *d* [E *e*]
		or	ES *w* . *d* [E *e*]
		or	G *w* . *d* [E *e*]
		or	L *w*
		or	A [*w*]
		or	D *w* . *d*
		or	DT [*char-literal-constant*] [(*v-list*)]
R1006	*w*	**is**	*int-literal-constant*
R1007	*m*	**is**	*int-literal-constant*
R1008	*d*	**is**	*int-literal-constant*
R1009	*e*	**is**	*int-literal-constant*
R1010	*v*	**is**	*signed-int-literal-constant*

C1005 (R1009) *e* shall be positive.

C1006 (R1006) *w* shall be zero or positive for the I, B, O, Z, and F edit descriptors. *w* shall be positive for all other edit descriptors.

C1007 (R1005) *w*, *m*, *d*, *e*, and *v* shall not have kind parameters specified for them.

C1008 (R1005) The *char-literal-constant* in the DT edit descriptor shall not have a kind parameter specified for it.

I, B, O, Z, F, E, EN, ES, G, L, A, D, and DT indicate the manner of editing.

R1011	*control-edit-desc*	**is**	*position-edit-desc*
		or	[*r*] /
		or	:
		or	*sign-edit-desc*
		or	*k* P
		or	*blank-interp-edit-desc*
		or	*round-edit-desc*
		or	*decimal-edit-desc*
R1012	*k*	**is**	*signed-int-literal-constant*

C1009 (R1012) *k* shall not have a kind parameter specified for it.

In *k* P, *k* is called the **scale factor**.

R1013	*position-edit-desc*	**is**	T *n*
		or	TL *n*
		or	TR *n*
		or	*n* X
R1014	*n*	**is**	*int-literal-constant*

C1010 (R1014) *n* shall be positive.

C1011 (R1014) *n* shall not have a kind parameter specified for it.

R1015	*sign-edit-desc*	**is**	SS
		or	SP

		or	S
R1016	*blank-interp-edit-desc*	**is**	BN
		or	BZ
R1017	*round-edit-desc*	**is**	RU
		or	RD
		or	RZ
		or	RN
		or	RC
		or	RP
R1018	*decimal-edit-desc*	**is**	DC
		or	DP

T, TL, TR, X, slash, colon, SS, SP, S, P, BN, BZ, RU, RD, RZ, RN, RC, RP, DC, and DP indicate the manner of editing.

R1019	*char-string-edit-desc*	**is**	*char-literal-constant*

C1012 (R1019) The *char-literal-constant* shall not have a kind parameter specified for it.

Each *rep-char* in a character string edit descriptor shall 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.

The edit descriptors are without regard to case except for the characters in the character constants.

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 start of formatted data transfer using a format specification initiates **format control** (9.5.3.4.2). Each action of format control depends on information jointly provided by

(1) The next edit descriptor in the format specification and
(2) The next effective item in the input/output list, if one exists.

If an input/output list specifies at least one effective list item, at least one data edit descriptor shall exist in the format specification.

NOTE 10.4

An empty format specification of the form ( ) may be used only if the input/output list has no effective list items (9.5.3.4). Zero length character items are effective list items, but zero sized arrays and implied DO lists with an iteration count of zero are not.

A format specification is interpreted from left to right. The exceptions are format items preceded by a repeat specification *r*, and format reversion (described below).

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 10.5

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 (9.5.2), except that an input/output list item of type complex requires the interpretation of two F, E, EN, ES, 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 in a manner identical to the way it is positioned when a slash edit descriptor is processed (10.7.2). Format control then reverts to the beginning of the format item terminated by the last preceding right parenthesis that is not part of a DT edit descriptor. If there is no such preceding right parenthesis, format control reverts to the first left parenthesis of the format specification. If any reversion occurs, the reused portion of the format specification shall 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 changeable modes (9.4.1).

NOTE 10.6

Example: The format specification:

```
10 FORMAT (1X, 2(F10.3, I5))
```

with an output list of

```
WRITE (10,10) 10.1, 3, 4.7, 1, 12.4, 5, 5.2, 6
```

produces the same output as the format specification:

```
10 FORMAT (1X, F10.3, I5, F10.3, I5/F10.3, I5, F10.3, I5)
```

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 in the current record.

After each T, TL, TR, or X edit descriptor is processed, the file is positioned as described in 10.7.1. After each slash edit descriptor is processed, the file is positioned as described in 10.7.2.

During formatted stream output, processing of an A edit descriptor may cause file positioning to occur (10.6.3).

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 (10.7.2).

During a read operation, any unprocessed characters of the current record are skipped whenever the next record is read.

10.5 Decimal symbol

The **decimal symbol** is the character that separates the whole and fractional parts in the decimal representation of a real number in an internal or external file. When the decimal edit mode is POINT, the decimal symbol is a decimal point. When the decimal edit mode is COMMA, the decimal symbol is a comma.

10.6 Data edit descriptors

Data edit descriptors cause the conversion of data to or from its internal representation; during formatted stream output, the A data edit descriptor may also cause file positioning. On input, the specified variable becomes defined unless an error condition, an end-of-file condition, or an end-of-record condition occurs. On output, the specified expression is evaluated.

During input from a Unicode file,

(1) characters in the record that correspond to an ASCII character variable shall have a position in the ISO 10646 character type collating sequence of 127 or less, and

(2) characters in the record that correspond to a default character variable shall be representable in the default character type.

During input from a non-Unicode file,

(1) characters in the record that correspond to a character variable shall have the kind of the character variable, and

(2) characters in the record that correspond to a numeric or logical variable shall be of default character type.

During output to a Unicode file, all characters transmitted to the record are of ISO 10646 character type. If a character input/output list item or character string edit descriptor contains a character that is not representable in the ISO 10646 character type, the result is processor-dependent.

During output to a non-Unicode file, characters transmitted to the record as a result of processing a character string edit descriptor or as a result of evaluating a numeric, logical, or default character data entity, are of type default character.

10.6.1 Numeric editing

The I, B, O, Z, F, E, EN, ES, D, and G edit descriptors may be used to specify the input/output of integer, real, and complex data. The following general rules apply:

(1) On input, leading blanks are not significant. When the input field is not an IEEE exceptional specification (10.6.1.2.1), the interpretation of blanks, other than leading blanks, is determined by the blank interpretation mode (10.7.6). Plus signs may be omitted. A field containing only blanks is considered to be zero.

(2) On input, with F, E, EN, ES, D, and G editing, a decimal symbol appearing in the input field overrides the portion of an edit descriptor that specifies the decimal symbol location. The input field may have more digits than the processor uses to approximate the value of the datum.

(3) On output with I, F, E, EN, ES, D, and G editing, the representation of a positive or zero internal value in the field may be prefixed with a plus sign, as controlled by the S, SP, and SS edit descriptors or the processor. The representation of a negative internal value in the field shall be prefixed with a minus sign.

(4) On output, the representation is right justified in the field. If the number of characters produced by the editing is smaller than the field width, leading blanks are inserted in the field.

(5) On output, if the number of characters produced exceeds the field width or if an exponent exceeds its specified length using the E*w.d* E*e*, EN*w.d* E*e*, ES*w.d* E*e*, or G*w.d* E*e* edit descriptor, the processor shall fill the entire field of width *w* with asterisks. However, the processor shall not produce asterisks if the field width is not exceeded when optional characters are omitted.

NOTE 10.7

When an SP edit descriptor is in effect, a plus sign is not optional.

(6) On output, with I, B, O, Z, and F editing, the specified value of the field width *w* may be zero. In such cases, the processor selects the smallest positive actual field width that does not result in a field filled with asterisks. The specified value of *w* shall not be zero on input.

10.6.1.1 Integer editing

The I*w*, I*w.m*, B*w*, B*w.m*, O*w*, O*w.m*, Z*w*, and Z*w.m* edit descriptors indicate that the field to be edited occupies *w* positions, except when *w* is zero. When *w* is zero, the processor selects the field width. On input, *w* shall not be zero. The specified input/output list item shall be of type integer. The G edit descriptor also may be used to edit integer data (10.6.4.1.1).

On input, *m* has no effect.

In the input field for the I edit descriptor, the character string shall be a *signed-digit-string* (R408), except for the interpretation of blanks. For the B, O, and Z edit descriptors, the character string shall consist of binary, octal, or hexadecimal digits (as in R412, R413, R414) in the respective input field. The lower-case hexadecimal digits a through f in a hexadecimal input field are equivalent to the corresponding upper-case hexadecimal digits.

The output field for the I*w* edit descriptor consists of zero or more leading blanks followed by a minus sign if the internal value is negative, or an optional plus sign otherwise, followed by the magnitude of the internal value as a *digit-string* without leading zeros.

NOTE 10.8

A *digit-string* always consists of at least one digit.

The output field for the B*w*, O*w*, and Z*w* descriptors consists of zero or more leading blanks followed by the internal value in a form identical to the digits of a binary, octal, or hexadecimal constant, respectively, with the same value and without leading zeros.

NOTE 10.9

A binary, octal, or hexadecimal constant always consists of at least one digit.

The output field for the I*w.m*, B*w.m*, O*w.m*, and Z*w.m* edit descriptor is the same as for the I*w*, B*w*, O*w*, and Z*w* edit descriptor, respectively, except that the *digit-string* consists of at least *m* digits. If necessary, sufficient leading zeros are included to achieve the minimum of *m* digits. The value of *m* shall not exceed the value of *w*, except when *w* is zero. If *m* is zero and the internal value is zero, the output field consists of only blank characters, regardless of the sign control in effect. When *m* and *w* are both zero, and the internal value is zero, one blank character is produced.

10.6.1.2 Real and complex editing

The F, E, EN, ES, and D edit descriptors specify the editing of real and complex data. An input/output list item corresponding to an F, E, EN, ES, or D edit descriptor shall be real or complex. The G edit descriptor also may be used to edit real and complex data (10.6.4.1.2).

A lower-case letter is equivalent to the corresponding upper-case letter in an IEEE exceptional specification or the exponent in a numeric input field.

10.6.1.2.1 F editing

The F*w.d* edit descriptor indicates that the field occupies *w* positions, the fractional part of which consists of *d* digits. When *w* is zero, the processor selects the field width. On input, *w* shall not be zero.

The input field is either an IEEE exceptional specification or consists of an optional sign, followed by a string of one or more digits optionally containing a decimal symbol, including any blanks interpreted as zeros. The *d* has no effect on input if the input field contains a decimal symbol. If the decimal symbol 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. The basic form may be followed by an exponent of one of the following forms:

(1) A *sign* followed by a *digit-string*

(2) E followed by zero or more blanks, followed by a *signed-digit-string*

(3) D followed by zero or more blanks, followed by a *signed-digit-string*

An exponent containing a D is processed identically to an exponent containing an E.

NOTE 10.10

If the input field does not contain an exponent, the effect is as if the basic form were followed by an exponent with a value of $-k$, where k is the established scale factor (10.7.5).

An input field that is an IEEE exceptional specification consists of optional blanks, followed by either of

(1) an optional sign, followed by the string 'INF' or the string 'INFINITY' or

(2) an optional sign, followed by the string 'NAN', optionally followed by zero or more alphanumeric characters enclosed in parentheses,

optionally followed by blanks.

The value specified by form (1) is an IEEE infinity; this form shall not be used if the processor does not support IEEE infinities for the input variable. The value specified by form (2) is an IEEE NaN; this form shall not be used if the processor does not support IEEE NaNs for the input variable. The NaN value is a quiet NaN if the only nonblank characters in the field are 'NAN' or 'NAN()'; otherwise, the NaN value is processor-dependent. The interpretation of a sign in a NaN input field is processor dependent.

For an internal value that is an IEEE infinity, the output field consists of blanks, if necessary, followed by a minus sign for negative infinity or an optional plus sign otherwise, followed by the letters 'Inf' or 'Infinity', right justified within the field. If w is less than 3, the field is filled with asterisks; otherwise, if w is less than 8, 'Inf' is produced.

For an internal value that is an IEEE NaN, the output field consists of blanks, if necessary, followed by the letters 'NaN' and optionally followed by one to $w-5$ alphanumeric processor-dependent characters enclosed in parentheses, right justified within the field. If w is less than 3, the field is filled with asterisks.

NOTE 10.11

> The processor-dependent characters following 'NaN' may convey additional information about that particular NaN.

For an internal value that is neither an IEEE infinity nor a NaN, the output field consists of blanks, if necessary, followed by a minus sign if the internal value is negative, or an optional plus sign otherwise, followed by a string of digits that contains a decimal symbol 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 symbol if the magnitude of the value in the output field is less than one. The optional zero shall appear if there would otherwise be no digits in the output field.

10.6.1.2.2 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*w.d* editing (10.6.1.2.1).

For an internal value that is an IEEE infinity or NaN, the form of the output field is the same as for F$w.d$.

For an internal value that is neither an IEEE infinity nor a NaN, the form of the output field for a scale factor of zero is:

$$[\ \pm\ ]\ [0].x_1x_2\ldots x_d exp$$

where:

$\pm$ signifies a plus sign or a minus sign.

. signifies a decimal symbol (10.5).

$x_1x_2\ldots x_d$ are the *d* most significant digits of the internal 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$\pm z_1z_2\ldots z_e$
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 each z is a digit.

The sign in the exponent is produced. A plus sign is produced if the exponent value is zero. The edit descriptor forms E*w.d* and D*w.d* shall not be used if |*exp*| > 999.

The scale factor *k* controls the decimal normalization (10.2.1, 10.7.5). If $-d < k \le 0$, the output field contains exactly $|k|$ leading zeros and $d-|k|$ significant digits after the decimal symbol. If $0 < k < d+2$, the output field contains exactly *k* significant digits to the left of the decimal symbol and $d-k+1$

significant digits to the right of the decimal symbol. Other values of *k* are not permitted.

10.6.1.2.3 EN editing

The EN edit descriptor produces an output field in the form of a real number in engineering notation such that the decimal exponent is divisible by three and the absolute value of the significand (R418) is greater than or equal to 1 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*w.d* editing (10.6.1.2.1).

For an internal value that is an IEEE infinity or NaN, the form of the output field is the same as for F$w.d$.

For an internal value that is neither an IEEE infinity nor a NaN, the form of the output field is:

[±] *yyy* . $x_1 x_2 \dots x_d$*exp*

where:

± signifies a plus sign or a minus sign.

yyy are the 1 to 3 decimal digits representative of the most significant digits of the internal value after rounding (*yyy* is an integer such that $1 \le yyy < 1000$ or, if the output value is zero, $yyy = 0$).

. signifies a decimal symbol (10.5).

$x_1 x_2 \dots x_d$ are the *d* next most significant digits of the internal value 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*	$\|exp\| \le 99$	E$\pm z_1 z_2$ or $\pm 0 z_1 z_2$
	$99 < \|exp\| \le 999$	$\pm z_1 z_2 z_3$
EN*w.d* E*e*	$\|exp\| \le 10^e - 1$	E$\pm z_1 z_2 \dots z_e$

where each z is a digit.

The sign in the exponent is produced. A plus sign is produced if the exponent value is zero. The edit descriptor form EN*w.d* shall not be used if $|exp| > 999$.

NOTE 10.12

Examples:

Internal Value	Output field Using SS, EN12.3
`6.421`	`6.421E+00`
`-.5`	`-500.000E-03`
`.00217`	`2.170E-03`
`4721.3`	`4.721E+03`

10.6.1.2.4 ES editing

The ES edit descriptor produces an output field in the form of a real number in scientific notation such that the absolute value of the significand (R418) is greater than or equal to 1 and less than 10, except when the output value is zero. The scale factor has no effect on output.

The forms of the edit descriptor are ES*w.d* and ES*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*w.d* editing (10.6.1.2.1).

For an internal value that is an IEEE infinity or NaN, the form of the output field is the same as for F$w.d$.

For an internal value that is neither an IEEE infinity nor a NaN, the form of the output field is:

[±] *y* . $x_1x_2 \ldots x_d$*exp*

where:

± signifies a plus sign or a minus sign.

y is a decimal digit representative of the most significant digit of the internal value after rounding.

. signifies a decimal symbol (10.5).

$x_1x_2 \ldots x_d$ are the *d* next most significant digits of the internal value after rounding.

exp is a decimal exponent having one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
ES*w.d*	$\lvert exp \rvert \leq 99$	E$\pm z_1z_2$ or $\pm 0z_1z_2$
	$99 < \lvert exp \rvert \leq 999$	$\pm z_1z_2z_3$
ES*w.d* E*e*	$\lvert exp \rvert \leq 10^e - 1$	E$\pm z_1z_2 \ldots z_e$

where each z is a digit.

The sign in the exponent is produced. A plus sign is produced if the exponent value is zero. The edit descriptor form ES*w.d* shall not be used if |*exp*| > 999.

NOTE 10.13

Examples:

Internal Value	Output field Using SS, ES12.3
`6.421`	`6.421E+00`
`-.5`	`-5.000E-01`
`.00217`	`2.170E-03`
`4721.3`	`4.721E+03`

10.6.1.2.5 Complex editing

A complex datum consists of a pair of separate real data. The editing of a scalar datum of complex type is specified by two edit descriptors each of which specifies the editing of real data. 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 edit descriptor for the real part and the edit descriptor for the imaginary part.

10.6.1.2.6 Rounding mode

The rounding mode can be specified by an OPEN statement (9.4.1), a data transfer input/output statement (9.5.1.12), or an edit descriptor (10.7.7).

In what follows, the term "decimal value" means the exact decimal number as given by the character string, while the term "internal value" means the number actually stored (typically in binary form) in the processor. For example, in dealing with the decimal constant 0.1, the decimal value is the exact mathematical quantity 1/10, which has no exact representation on most processors. Formatted output of real data involves conversion from an internal value to a decimal value; formatted input involves conversion from a decimal value to an internal value.

When the I/O rounding mode is UP, the value resulting from conversion shall be the smallest representable value that is greater than or equal to the original value. When the I/O rounding mode is DOWN, the value resulting from conversion shall be the largest representable value that is less than or equal to the original value. When the I/O rounding mode is ZERO, the value resulting from conversion shall be the value closest to the original value and no greater in magnitude than the original value. When the I/O rounding mode is NEAREST, the value resulting from conversion shall be the closer of the two nearest representable values if one is closer than the other. If the two nearest representable values are equidistant from the original value, it is processor dependent which one of them is chosen. When the I/O rounding mode is COMPATIBLE, the value resulting from conversion shall be the closer of the two nearest representable values or the value away from zero if halfway between them. When the I/O rounding mode is PROCESSOR_DEFINED, rounding during conversion shall be a processor dependent default mode, which may correspond to one of the other modes.

On processors that support IEEE rounding on conversions, NEAREST shall correspond to round to nearest, as specified in the IEEE standard.

NOTE 10.14

On processors that support IEEE rounding on conversions, the I/O rounding modes COMPATIBLE and NEAREST will produce the same results except when the datum is halfway between the two representable values. In that case, NEAREST will pick the even value, but COMPATIBLE will pick the value away from zero. The I/O rounding modes UP, DOWN, and ZERO have the same effect as those specified in the IEEE standard for round toward $+\infty$, round toward $-\infty$, and round toward 0, respectively.

10.6.2 Logical editing

The L*w* edit descriptor indicates that the field occupies *w* positions. The specified input/output list item shall be of type logical. The G edit descriptor also may be used to edit logical data (10.6.4.2).

The input field consists of optional blanks, optionally followed by a period, followed by a T for true or F for false. The T or F may be followed by additional characters in the field, which are ignored.

A lower-case letter is equivalent to the corresponding upper-case letter in a logical input field.

NOTE 10.15

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 internal value is true or false, respectively.

10.6.3 Character editing

The A[*w*] edit descriptor is used with an input/output list item of type character. The G edit descriptor also may be used to edit character data (10.6.4.3). The kind type parameter of all characters transferred and converted under control of one A or G edit descriptor is implied by the kind of the corresponding list item.

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 corresponding list item, regardless of the value of the kind type parameter.

Let *len* be the length of the input/output list item. If the specified field width *w* for an A edit descriptor corresponding to an input item is greater than or equal to *len*, the rightmost *len* characters will be taken from the input field. If the specified field width *w* is less than *len*, the *w* characters will appear left justified with *len*$-w$ trailing blanks in the internal value.

If the specified field width *w* for an A edit descriptor corresponding to an output item is greater than *len*, the output field will consist of $w - len$ blanks followed by the *len* characters from the internal value. 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 value.

NOTE 10.16

For nondefault character types, the blank padding character is processor dependent.

If the file is connected for stream access, the output may be split across more than one record if it contains newline characters. A newline character is a nonblank character returned by the intrinsic function NEW_LINE. Beginning with the first character of the output field, each character that is not a newline is written to the current record in successive positions; each newline character causes file positioning at that point as if by slash editing (the current record is terminated at that point, a new empty record is created following the current record, this new record becomes the last and current record of the file, and the file is positioned at the beginning of this new record).

NOTE 10.17

If the intrinsic function NEW_LINE returns a blank character for a particular character kind, then the processor does not support using a character of that kind to cause record termination in a formatted stream file.

10.6.4 Generalized editing

The G*w.d* and G*w.d* E*e* edit descriptors are used with an input/output list item of any intrinsic type. These edit descriptors indicate that the external field occupies *w* positions, the fractional part of which consists of a maximum of *d* digits and the exponent part consists of *e* digits. When these edit descriptors are used to specify the input/output of integer, logical, or character data, *d* and *e* have no effect.

10.6.4.1 Generalized numeric editing

When used to specify the input/output of integer, real, and complex data, the G*w.d* and G*w.d* E*e* edit descriptors follow the general rules for numeric editing (10.6.1).

NOTE 10.18

The G*w.d* E*e* edit descriptor follows any additional rules for the E*w.d* E*e* edit descriptor.

10.6.4.1.1 Generalized integer editing

When used to specify the input/output of integer data, the G*w.d* and G*w.d* E*e* edit descriptors follow the rules for the I*w* edit descriptor (10.6.1.1), except that *w* shall not be zero.

10.6.4.1.2 Generalized real and complex editing

The form and interpretation of the input field is the same as for F*w.d* editing (10.6.1.2.1).

For an internal value that is an IEEE infinity or NaN, the form of the output field is the same as for F$w.d$.

Otherwise, the method of representation in the output field depends on the magnitude of the internal value being edited. Let N be the magnitude of the internal value and r be the rounding mode value defined in the table below. If $0 < N < 0.1 - r \times 10^{-d-1}$ or $N \geq 10^d - r$, or N is identically 0 and *d* is 0, 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 (10.7.5) currently in effect. If $0.1 - r \times 10^{-d-1} \leq N < 10^d - r$ or N is identically 0 and *d* is not zero, the scale factor has no effect, and the value of N determines the editing as follows:

Magnitude of Internal Value	Equivalent Conversion
$N = 0$	$\mathrm{F}(w-n).(d-1)$, n('b')
$0.1 - r \times 10^{-d-1} \leq N < 1 - r \times 10^{-d}$	$\mathrm{F}(w-n).d$, n('b')
$1 - r \times 10^{-d} \leq N < 10 - r \times 10^{-d+1}$	$\mathrm{F}(w-n).(d-1)$, n('b')
$10 - r \times 10^{-d+1} \leq N < 100 - r \times 10^{-d+2}$	$\mathrm{F}(w-n).(d-2)$, n('b')
.	.
.	.
.	.
$10^{d-2} - r \times 10^{-2} \leq N < 10^{d-1} - r \times 10^{-1}$	$\mathrm{F}(w-n).1$, n('b')
$10^{d-1} - r \times 10^{-1} \leq N < 10^d - r$	$\mathrm{F}(w-n).0$, n('b')

where b is a blank, n is 4 for G*w.d* and *e* + 2 for G*w.d* E*e*, and r is defined for each rounding mode as follows:

I/O Rounding Mode	r
COMPATIBLE	0.5
NEAREST	0.5 if the higher value is even −0.5 if the lower value is even
UP	1
DOWN	0
ZERO	1 if internal value is negative 0 if internal value is positive

The value of $w - n$ shall be positive

NOTE 10.19

The scale factor has no effect on output unless the magnitude of the datum to be edited is outside the range that permits effective use of F editing.

10.6.4.2 Generalized logical editing

When used to specify the input/output of logical data, the G*w.d* and G*w.d* E*e* edit descriptors follow the rules for the L*w* edit descriptor (10.6.2).

10.6.4.3 Generalized character editing

When used to specify the input/output of character data, the G*w.d* and G*w.d* E*e* edit descriptors follow the rules for the A*w* edit descriptor (10.6.3).

10.6.5 User-defined derived-type editing

The DT edit descriptor allows a user-provided procedure to be used instead of the processor's default input/output formatting for processing a list item of derived type.

The DT edit descriptor may include a character literal constant. The character value "DT" concatenated with the character literal constant is passed to the user-defined derived-type input/output procedure as the `iotype` argument (9.5.3.7). The *v* values of the edit descriptor are passed to the user-defined derived-type input/output procedure as the `v_list` array argument.

NOTE 10.20

For the edit descriptor `DT'Link List'(10, 4, 2)`, `iotype` is `"DTLink List"` and `v_list` is `(/10, 4, 2/)`.

If a derived-type variable or value corresponds to a DT edit descriptor, there shall be an accessible interface to a corresponding derived-type input/output procedure for that derived type (9.5.3.7). A DT edit descriptor shall not correspond with a list item that is not of a derived type.

10.7 Control edit descriptors

A control edit descriptor does not cause the transfer of data or the conversion of data to or from internal representation, but may affect the conversions performed by subsequent data edit descriptors.

10.7.1 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. If any character skipped by a T, TL, TR, or X edit descriptor is of type nondefault character, and the unit is an internal file of type default character or an external non-Unicode file, the result of that position editing is processor dependent.

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 10.21

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.

10.7.1.1 T, TL, and TR editing

The **left tab limit** affects file positioning by the T and TL edit descriptors. Immediately prior to data transfer, the left tab limit becomes defined as the character position of the current record or the current position of the stream file. If, during data transfer, the file is positioned to another record, the left tab limit becomes defined as character position one of that record.

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 of the record, relative to the left tab limit.

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 *n* is greater than the difference between the current position and the left tab limit, the TL*n* edit descriptor indicates that the transmission of the next character to or from the record is to occur at the left tab limit.

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 10.22

The *n* in a T*n*, TL*n*, or TR*n* edit descriptor shall be specified and shall be greater than zero.

10.7.1.2 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 10.23

The *n* in an *n*X edit descriptor shall be specified and shall be greater than zero.

10.7.2 Slash editing

The slash edit descriptor indicates the end of data transfer to or from the current record.

On input from a file connected for sequential or stream 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 or stream access, a new empty record is created following the current record; this new record then becomes the last and current record of the file and the file is positioned at the beginning of this new record.

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, if there is such a record, and this record becomes the current record.

NOTE 10.24

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.

An entire record may be skipped on input.

The repeat specification is optional in the slash edit descriptor. If it is not specified, the default value is one.

10.7.3 Colon editing

The colon edit descriptor terminates format control if there are no more effective items in the input/output list (9.5.2). The colon edit descriptor has no effect if there are more effective items in the input/output list.

10.7.4 SS, SP, and S editing

The SS, SP, and S edit descriptors temporarily change (9.4.1) the sign mode (9.4.5.14, 9.5.1.13) for the connection. The edit descriptors SS, SP, and S set the sign mode corresponding to the SIGN= specifier values SUPPRESS, PLUS, and PROCESSOR_DEFINED, respectively.

The sign mode controls optional plus characters in numeric output fields. When the sign mode is PLUS, the processor shall produce a plus sign in any position that normally contains an optional plus sign. When the sign mode is SUPPRESS, the processor shall not produce a plus sign in such positions. When the sign mode is PROCESSOR_DEFINED, the processor has the option of producing a plus sign or not in such positions, subject to 10.6.1(5).

The SS, SP, and S edit descriptors affect only I, F, E, EN, ES, D, and G editing during the execution of an output statement. The SS, SP, and S edit descriptors have no effect during the execution of an input statement.

10.7.5 P editing

The kP edit descriptor temporarily changes (9.4.1) the scale factor for the connection to k. The scale factor affects the editing of F, E, EN, ES, D, and G edit descriptors for numeric quantities.

The scale factor k affects the appropriate editing in the following manner:

(1) On input, with F, E, EN, ES, 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, ES, 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 (R418) 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 and ES editing, the scale factor has no effect.

If UP, DOWN, ZERO, or NEAREST I/O rounding mode is in effect then

(1) On input, the scale factor is applied to the external decimal value and then this is converted using the current I/O rounding mode, and

(2) On output, the internal value is converted using the current I/O rounding mode and then the scale factor is applied to the converted decimal value.

10.7.6 BN and BZ editing

The BN and BZ edit descriptors temporarily change (9.4.1) the blank interpretation mode (9.4.5.4, 9.5.1.5) for the connection. The edit descriptors BN and BZ set the blank interpretation mode corresponding to the BLANK= specifier values NULL and ZERO, respectively.

The blank interpretation mode controls the interpretation of nonleading blanks in numeric input fields. Such blank characters are interpreted as zeros when the blank interpretation mode has the value ZERO; they are ignored when the blank interpretation mode has the value NULL. 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.

The blank interpretation mode affects only numeric editing (10.6.1) and generalized numeric editing (10.6.4.1) on input. It has no effect on output.

10.7.7 RU, RD, RZ, RN, RC, and RP editing

The round edit descriptors temporarily change (9.4.1) the connection's I/O rounding mode (9.4.5.13, 9.5.1.12, 10.6.1.2.6). The round edit descriptors RU, RD, RZ, RN, RC, and RP set the I/O rounding mode corresponding to the ROUND= specifier values UP, DOWN, ZERO, NEAREST, COMPATIBLE, and PROCESSOR_DEFINED, respectively. The I/O rounding mode affects the conversion of real and complex values in formatted input/output. It affects only D, E, EN, ES, F, and G editing.

10.7.8 DC and DP editing

The decimal edit descriptors temporarily change (9.4.1) the decimal edit mode (9.4.5.5, 9.5.1.6) for the connection. The edit descriptors DC and DP set the decimal edit mode corresponding to the DECIMAL= specifier values COMMA and POINT, respectively.

The decimal edit mode controls the representation of the decimal symbol (10.5) during conversion of real and complex values in formatted input/output. The decimal edit mode affects only D, E, EN, ES, F, and G editing. If the mode is COMMA during list-directed input/output, the character used as a value separator is a semicolon in place of a comma.

10.8 Character string edit descriptors

A character string edit descriptor shall not be used on input.

The character string edit descriptor causes characters to be written from the enclosed characters of the edit descriptor itself, including blanks. For a character string edit descriptor, the width of the field is the number of characters between the delimiting characters. Within the field, two consecutive delimiting characters are counted as a single character.

NOTE 10.25

A delimiter for a character string edit descriptor is either an apostrophe or quote.

10.9 List-directed formatting

List-directed input/output allows data editing according to the type of the list item instead of by a format specification. It also allows data to be free-field, that is, separated by commas (or semicolons) or blanks.

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 null value or one of the forms:

c
*r*c*
*r**

where *c* is a literal constant, optionally signed if integer or real, or an undelimited character constant and *r* is an unsigned, nonzero, integer literal constant. Neither *c* nor *r* shall have kind type parameters specified. The constant *c* is interpreted as though it had the same kind type parameter as the corresponding list item. 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

(1) A comma optionally preceded by one or more contiguous blanks and optionally followed by one or more contiguous blanks, unless the decimal edit mode is COMMA, in which case a semicolon is used in place of the comma,

(2) A slash optionally preceded by one or more contiguous blanks and optionally followed by one or more contiguous blanks, or

(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.

NOTE 10.26

Although a slash encountered in an input record is referred to as a separator, it actually causes termination of list-directed and namelist input statements; it does not actually separate two values.

NOTE 10.27

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.

10.9.1 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 shall be acceptable for the type of the next effective item in the list. Blanks are never used as zeros, and embedded blanks are not permitted in constants, except within character constants and complex constants as specified below.

For the *r***c* form of an input value, the constant *c* is interpreted as an undelimited character constant if the first list item corresponding to this value is of type default, ASCII, or ISO 10646 character, there is a nonblank character immediately after *r**, and that character is not an apostrophe or a quotation mark; otherwise, *c* is interpreted as a literal constant.

NOTE 10.28

The end of a record has the effect of a blank, except when it appears within a character constant.

When the next effective item is of type integer, the value in the input record is interpreted as if an I*w* edit descriptor with a suitable value of *w* were used.

When the next effective item is of type real, the input form is that of a numeric input field. A numeric input field is a field suitable for F editing (10.6.1.2.1) that is assumed to have no fractional digits unless a decimal symbol appears within the field.

When the next effective 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 separator, 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 any number of blanks and ends of records. The separator is a comma if the decimal edit mode is POINT; it is a semicolon if the decimal edit mode is COMMA. The end of a record may occur between the real part and the separator or between the separator and the imaginary part.

When the next effective item is of type logical, the input form shall not include value separators among the optional characters permitted for L editing.

When the next effective item is of type character, the input form consists of a possibly delimited sequence of zero or more *rep-char*s whose kind type parameter is implied by the kind of the effective list item. Character sequences may be continued from the end of one record to the beginning of the next record, but the end of record shall not occur between a doubled apostrophe in an apostrophe-delimited character sequence, nor between a doubled quote in a quote-delimited character sequence. The end of the record does not cause a blank or any other character to become part of the character sequence. The character sequence may be continued on as many records as needed. The characters blank, comma, semicolon, and slash may appear in default, ASCII, or ISO 10646 character sequences.

If the next effective item is of type default, ASCII, or ISO 10646 character and

(1) The character sequence does not contain value separators,
(2) The character sequence does not cross a record boundary,
(3) The first nonblank character is not a quotation mark or an apostrophe,
(4) The leading characters are not *digit*s followed by an asterisk, and
(5) The character sequence contains at least one character,

the delimiting apostrophes or quotation marks are not required. If the delimiters are omitted, the character sequence is terminated by the first blank, comma, slash, or end of record; in this case apostrophes and quotation marks within the datum are not to be doubled.

Let *len* be the length of the next effective item, and let w be the length of the character sequence. If *len* is less than or equal to w, the leftmost *len* characters of the sequence are transmitted to the next effective item. If *len* is greater than w, the sequence is transmitted to the leftmost w characters of the next effective item and the remaining *len*$-w$ characters of the next effective item are filled with blanks. The effect is as though the sequence were assigned to the next effective item in a character assignment statement (7.4.1.3).

10.9.1.1 Null values

A null value is specified by

(1) The *r** form,
(2) No characters between consecutive value separators, or
(3) No characters before the first value separator in the first record read by each execution of a list-directed input statement.

NOTE 10.29

The end of a record following any other value separator, with or without separating blanks, does not specify a null value in list-directed input.

A null value has no effect on the definition status of the next effective item. A null value shall not be used for either the real or imaginary part of a complex constant, but a single null value may represent an entire complex constant.

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. Any characters remaining in the current record are ignored. 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 10.30

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 sequence
(2) Embedded blanks surrounding the real or imaginary part of a complex constant

NOTE 10.30 (cont.)

(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

NOTE 10.31

List-directed input example:

```
INTEGER I; REAL X (8); CHARACTER (11) P;
COMPLEX Z; LOGICAL G
  ...
READ *, I, X, P, Z, G
  ...
```

The input data records are:

```
12345,12345,,2*1.5,4*
ISN'T_BOB'S,(123,0),.TEXAS$
```

The results are:

Variable	Value
I	12345
X (1)	12345.0
X (2)	unchanged
X (3)	1.5
X (4)	1.5
X (5) – X (8)	unchanged
P	ISN'T_BOB'S
Z	(123.0,0.0)
G	true

10.9.2 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 adjacent undelimited character sequences, the values are separated by one or more blanks or by a comma, or a semicolon if the decimal edit mode is 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 the end of record shall not occur within a constant except for complex constants and character sequences. The processor shall not insert blanks within character sequences or within constants, except for complex constants.

Logical output values are T for the value true and F for the value false.

Integer output constants are produced with the effect of an I*w* edit descriptor.

Real 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 $10^{d_1} \leq x < 10^{d_2}$, where d_1 and d_2 are processor-dependent integers. If the magnitude *x* is within this range or is zero, the constant is produced using 0PF*w.d*; otherwise, 1PE*w.d* E*e* is used.

For numeric output, reasonable processor-dependent values of *w*, *d*, and *e* are used for each of the numeric constants output.

Complex constants are enclosed in parentheses with a separator between the real and imaginary parts, each produced as defined above for real constants. The separator is a comma if the decimal edit mode is POINT; it is a semicolon if the decimal edit mode is COMMA. The end of a record may occur between the separator 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 separator and the end of a record and one blank at the beginning of the next record.

Character sequences produced when the delimiter mode has a value of NONE

(1) Are not delimited by apostrophes or quotation marks,
(2) Are not separated from each other by value separators,
(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 at the beginning of any record that begins with the continuation of a character sequence from the preceding record.

Character sequences produced when the delimiter mode has 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 contiguous quotes.

Character sequences produced when the delimiter mode has 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 contiguous 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 as output by list-directed formatting.

Except for continuation of delimited character sequences, each output record begins with a blank character.

NOTE 10.32

The length of the output records is not specified exactly and may be processor dependent.

10.10 Namelist formatting

Namelist input/output allows data editing with NAME=value subsequences. This facilitates documentation of input and output files and more flexibility on input.

The characters in one or more namelist records constitute a sequence of **name-value subsequences**, each of which consists of an object designator followed by an equals and followed by one or more values and value separators. The equals may optionally be preceded or followed by 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 *namelist-group-object-list* (5.4).

Each value is either a null value (10.10.1.4) or one of the forms

c
*r***c*
*r**

where *c* is a literal constant, optionally signed if integer or real, and *r* is an unsigned, nonzero, integer literal constant. Neither *c* nor *r* may have kind type parameters specified. The constant *c* is interpreted as though it had the same kind type parameter as the corresponding list item. 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 namelist formatting is the same as for list-directed formatting (10.9).

10.10.1 Namelist input

Input for a namelist input statement consists of

(1) Optional blanks and namelist comments,
(2) The character & followed immediately by the *namelist-group-name* as specified in the NAMELIST statement,
(3) One or more blanks,
(4) A sequence of zero or more name-value subsequences separated by value separators, and
(5) A slash to terminate the namelist input.

NOTE 10.33

A slash encountered in a namelist input record causes the input statement to terminate. A slash cannot be used to separate two values in a namelist input statement.

In each name-value subsequence, the name shall be the name of a namelist group object list item with an optional qualification and the name with the optional qualification shall not be a zero-sized array, a zero-sized array section, or a zero-length character string. The optional qualification, if any, shall not contain a vector subscript.

A group name or object name is without regard to case.

10.10.1.1 Namelist group object names

Within the input data, each name shall correspond to a particular namelist group object name. Subscripts, strides, and substring range expressions used to qualify group object names shall be optionally signed integer literal constants with no kind type parameters specified. If a namelist group object is an array, the input record corresponding to it may contain either the array name or the designator of a subobject of that array, using the syntax of object designators (R603). If the namelist group object 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 the designator 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 namelist group object items. The input records need not contain all the names of the namelist group object items. The definition status of any names from the *namelist-group-object-list* that do not occur in the input record remains unchanged. In the input record, each object name or subobject designator may be preceded and followed by one or more optional blanks but shall not contain embedded blanks.

10.10.1.2 Namelist input values

The datum *c* (10.10) is any input value acceptable to format specifications for a given type, except for a restriction on the form of input values corresponding to list items of types logical, integer, and character as specified in 10.10.1.3. The form of a real or complex value is dependent on the decimal edit mode in effect (10.7.8). The form of an input value shall be acceptable for the type of the namelist group object list item. The number and forms of the input values that 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 that of a scalar variable of an intrinsic type, the equals shall not be followed by more than one value. Blanks are never used as zeros, and embedded blanks are not permitted in constants except within character constants and complex constants as specified in 10.10.1.3.

The name-value subsequences are evaluated serially, in left-to-right order. A namelist group object designator may appear in more than one name-value sequence.

end of record shall not occur between a doubled apostrophe in an apostrophe-delimited sequence, nor between a doubled quote in a quote-delimited sequence. The end of the record does not cause a blank or any other character to become part of the sequence. The sequence may be continued on as many records as needed. The characters blank, comma, and slash may appear in such character sequences.

NOTE 10.35

A character sequence corresponding to a namelist input item of character type shall be delimited either with apostrophes or with quotes. The delimiter is required to avoid ambiguity between undelimited character sequences and object names. The value of the DELIM= specifier, if any, in the OPEN statement for an external file is ignored during namelist input (9.4.5.6).

Let *len* be the length of the next effective item, and let w be the length of the character sequence. If *len* is less than or equal to w, the leftmost *len* characters of the sequence are transmitted to the next effective item. If *len* is greater than w, the constant is transmitted to the leftmost w characters of the next effective item and the remaining *len*$-w$ characters of the next effective item are filled with blanks. The effect is as though the sequence were assigned to the next effective item in a character assignment statement (7.4.1.3).

10.10.1.4 Null values

A null value is specified by

(1) The *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 namelist group object list item is defined, it retains its previous value; if it is undefined, it remains undefined. A null value shall 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 10.36

The end of a record following a value separator, with or without intervening blanks, does not specify a null value in namelist input.

10.10.1.5 Blanks

All blanks in a namelist 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, or a semicolon if the decimal edit mode is comma, and
(4) Blanks between a name and the following equals.

10.10.1.6 Namelist Comments

Except within a character literal constant, a "!" character after a value separator or in the first nonblank position of a namelist input record initiates a comment. The comment extends to the end of the current input record and may contain any graphic character in the processor-dependent character set. The comment is ignored. A slash within the namelist comment does not terminate execution of the namelist input statement. Namelist comments are not allowed in stream input because comments depend on record structure.

NOTE 10.37

Namelist input example:

```
INTEGER I; REAL X (8); CHARACTER (11) P; COMPLEX Z;
LOGICAL G
NAMELIST / TODAY / G, I, P, Z, X
READ (*, NML = TODAY)
```

The input data records are:

```
&TODAY I = 12345, X(1) = 12345, X(3:4) = 2*1.5, I=6, ! This is a comment.
P = ''ISN'T_BOB'S'', Z = (123,0)/
```

The results stored are:

Variable	Value
I	6
X (1)	12345.0
X (2)	unchanged
X (3)	1.5
X (4)	1.5
X (5) – X (8)	unchanged
P	ISN'T_BOB'S
Z	(123.0,0.0)
G	unchanged

10.10.2 Namelist output

The form of the output produced is the same as that required for input, except for the forms of real, character, and logical values. The name in the output is in upper case. With the exception of adjacent undelimited character values, the values are separated by one or more blanks or by a comma, or a semicolon if the decimal edit mode is COMMA, optionally preceded by one or more blanks and optionally followed by one or more blanks.

Namelist output shall not include namelist comments.

The processor may begin new records as necessary. However, except for complex constants and character values, the end of a record shall not occur within a constant, character value, or name, and blanks shall not appear within a constant, character value, or name.

NOTE 10.38

The length of the output records is not specified exactly and may be processor dependent.

10.10.2.1 Namelist output editing

Logical output values are T for the value true and F for the value false.

Integer output constants are produced with the effect of an I*w* edit descriptor.

Real 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 $10^{d_1} \leq x < 10^{d_2}$, where d_1 and d_2 are processor-dependent integers. If the magnitude x is within this range or is zero, 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 numeric constants output.

Complex constants are enclosed in parentheses with a separator between the real and imaginary parts, each produced as defined above for real constants. The separator is a comma if the decimal edit mode is POINT; it is a semicolon if the decimal edit mode is COMMA. The end of a record may occur between the separator 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 separator and the end of a record and one blank at the beginning of the next record.

Character sequences produced when the delimiter mode has a value of NONE

(1) Are not delimited by apostrophes or quotation marks,
(2) Are not separated from each other by value separators,
(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 at the beginning of any record that begins with the continuation of a character sequence from the preceding record.

NOTE 10.39

Namelist output records produced with a DELIM= specifier with a value of NONE and which contain a character sequence might not be acceptable as namelist input records.

Character sequences produced when the delimiter mode has 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 contiguous quotes.

Character sequences produced when the delimiter mode has 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 contiguous apostrophes.

10.10.2.2 Namelist output records

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 namelist group object list item is placed in the output record followed by an equals and a list of values of the namelist group object list item.

An ampersand character followed immediately by a *namelist-group-name* will be produced by namelist formatting at the start of the first output record to indicate which particular group of data objects is being output. A slash is produced by namelist formatting to indicate the end of the namelist formatting.

A null value is not produced by namelist formatting.

Except for continuation of delimited character sequences, each output record begins with a blank character.

Section 11: Program units

The terms and basic concepts of program units were introduced in 2.2. A program unit is a main program, an external subprogram, a module, or a block data program unit.

This section describes main programs, modules, and block data program units. Section 12 describes external subprograms.

11.1 Main program

A Fortran **main program** is a program unit that does not contain a SUBROUTINE, FUNCTION, MODULE, or BLOCK DATA statement as its first statement.

R1101	*main-program*	**is**	[*program-stmt*]
			[*specification-part*]
			[*execution-part*]
			[*internal-subprogram-part*]
			end-program-stmt
R1102	*program-stmt*	**is**	PROGRAM *program-name*
R1103	*end-program-stmt*	**is**	END [PROGRAM [*program-name*]]

C1101 (R1101) In a *main-program*, the *execution-part* shall not contain a RETURN statement or an ENTRY statement.

C1102 (R1101) The *program-name* may be included in the *end-program-stmt* only if the optional *program-stmt* is used and, if included, shall be identical to the *program-name* specified in the *program-stmt*.

C1103 (R1101) An automatic object shall not appear in the *specification-part* (R204) of a main program.

NOTE 11.1

The program name is global to the program (16.1). For explanatory information about uses for the program name, see section C.8.1.

NOTE 11.2

An example of a main program is:

```
PROGRAM ANALYZE
   REAL A, B, C (10,10)     !  Specification part
   CALL FIND                !  Execution part
CONTAINS
   SUBROUTINE FIND          !  Internal subprogram
       ...
   END SUBROUTINE FIND
END PROGRAM ANALYZE
```

The main program may be defined by means other than Fortran; in that case, the program shall not contain a *main-program* program unit.

A reference to a Fortran *main-program* shall not appear in any program unit in the program, including itself.

11.2 Modules

A **module** contains specifications and definitions that are to be accessible to other program units. A module that is provided as an inherent part of the processor is an **intrinsic module**. A **nonintrinsic module** is defined by a module program unit or a means other than Fortran.

Procedures and types defined in an intrinsic module are not themselves intrinsic.

R1104 *module* **is** *module-stmt*
  [*specification-part*]
  [*module-subprogram-part*]
  end-module-stmt
R1105 *module-stmt* **is** MODULE *module-name*
R1106 *end-module-stmt* **is** END [MODULE [*module-name*]]
R1107 *module-subprogram-part* **is** *contains-stmt*
  module-subprogram
  [*module-subprogram*] ...
R1108 *module-subprogram* **is** *function-subprogram*
  or *subroutine-subprogram*

C1104 (R1104) If the *module-name* is specified in the *end-module-stmt*, it shall be identical to the *module-name* specified in the *module-stmt*.

C1105 (R1104) A module *specification-part* shall not contain a *stmt-function-stmt*, an *entry-stmt*, or a *format-stmt*.

C1106 (R1104) An automatic object shall not appear in the *specification-part* of a module.

C1107 (R1104) If an object of a type for which *component-initialization* is specified (R444) appears in the *specification-part* of a module and does not have the ALLOCATABLE or POINTER attribute, the object shall have the SAVE attribute.

NOTE 11.3

The module name is global to the program (16.1).

NOTE 11.4

Although statement function definitions, ENTRY statements, and FORMAT statements shall not appear in the specification part of a module, they may appear in the specification part of a module subprogram in the module.

A module is host to any module subprograms (12.1.2.2) it contains, and the entities in the module are therefore accessible in the module subprograms through host association.

NOTE 11.5

For a discussion of the impact of modules on dependent compilation, see section C.8.2.

NOTE 11.6

For examples of the use of modules, see section C.8.3.

If a procedure declared in the scoping unit of a module has an implicit interface, it shall be given the

EXTERNAL attribute in that scoping unit; if it is a function, its type and type parameters shall be explicitly declared in a type declaration statement in that scoping unit.

If an intrinsic procedure is declared in the scoping unit of a module, it shall explicitly be given the INTRINSIC attribute in that scoping unit or be used as an intrinsic procedure in that scoping unit.

11.2.1 The USE statement and use association

The **USE statement** specifies use association. A USE statement is a **module reference** to the module it specifies. At the time a USE statement is processed, the public portions of the specified module shall be available. A module shall not reference itself, either directly or indirectly.

The **USE statement** provides the means by which a scoping unit accesses named data objects, derived types, interface blocks, procedures, abstract interfaces, generic identifiers (12.3.2.1), and namelist groups in a module. The entities in the scoping unit are said to be **use associated** with the entities in the module. The accessed entities have the attributes specified in the module. The entities made accessible are identified by the names or generic identifiers used to identify them in the module. By default, they are identified by the same identifiers in the scoping unit containing the USE statement, but it is possible to specify that different local identifiers are used.

NOTE 11.7

The accessibility of module entities may be controlled by accessibility attributes (4.5.1.1, 5.1.2.1), and the ONLY option of the USE statement. Definability of module entities can be controlled by the PROTECTED attribute (5.1.2.12).

R1109	*use-stmt*	**is**	USE [[, *module-nature*] ::] *module-name* [, *rename-list*]
		or	USE [[, *module-nature*] ::] *module-name* , ■
			■ ONLY : [*only-list*]
R1110	*module-nature*	**is**	INTRINSIC
		or	NON_INTRINSIC
R1111	*rename*	**is**	*local-name* => *use-name*
		or	OPERATOR (*local-defined-operator*) => ■
			■ OPERATOR (*use-defined-operator*)
R1112	*only*	**is**	*generic-spec*
		or	*only-use-name*
		or	*rename*
R1113	*only-use-name*	**is**	*use-name*

C1108 (R1109) If *module-nature* is INTRINSIC, *module-name* shall be the name of an intrinsic module.

C1109 (R1109) If *module-nature* is NON_INTRINSIC, *module-name* shall be the name of a nonintrinsic module.

C1110 (R1109) A scoping unit shall not access an intrinsic module and a nonintrinsic module of the same name.

C1111 (R1111) OPERATOR(*use-defined-operator*) shall not identify a *generic-binding*.

C1112 (R1112) The *generic-spec* shall not identify a *generic-binding*.

NOTE 11.8

> The above two constraints do not prevent accessing a *generic-spec* that is declared by an interface block, even if a *generic-binding* has the same *generic-spec*.

C1113 (R1112) Each *generic-spec* shall be a public entity in the module.

C1114 (R1113) Each *use-name* shall be the name of a public entity in the module.

R1114	*local-defined-operator*	**is**	*defined-unary-op*
		or	*defined-binary-op*
R1115	*use-defined-operator*	**is**	*defined-unary-op*
		or	*defined-binary-op*

C1115 (R1115) Each *use-defined-operator* shall be a public entity in the module.

A *use-stmt* without a *module-nature* provides access either to an intrinsic or to a nonintrinsic module. If the *module-name* is the name of both an intrinsic and a nonintrinsic module, the nonintrinsic module is accessed.

The USE statement without the ONLY option provides access to all public entities in the specified module.

A USE statement with the ONLY option provides access only to those entities that appear as *generic-spec*s, *use-name*s, or *use-defined-operator*s in the *only-list*.

More than one USE statement for a given module may appear in a scoping unit. If one of the USE statements is without an ONLY qualifier, all public entities in the module are accessible. If all the USE statements have ONLY qualifiers, only those entities in one or more of the *only-list*s are accessible.

An accessible entity in the referenced module has one or more local identifiers. These identifiers are

(1) The identifier of the entity in the referenced module if that identifier appears as an *only-use-name* or as the *defined-operator* of a *generic-spec* in any *only* for that module,

(2) Each of the *local-name*s or *local-defined-operator*s that the entity is given in any *rename* for that module, and

(3) The identifier of the entity in the referenced module if that identifier does not appear as a *use-name* or *use-defined-operator* in any *rename* for that module.

Two or more accessible entities, other than generic interfaces or defined operators, may have the same identifier only if the identifier is not used to refer to an entity in the scoping unit. Generic interfaces and defined operators are handled as described in section 16.2.3. Except for these cases, the local identifier of any entity given accessibility by a USE statement shall differ from the local identifiers of all other entities accessible to the scoping unit through USE statements and otherwise.

NOTE 11.9

> There is no prohibition against a *use-name* or *use-defined-operator* appearing multiple times in one USE statement or in multiple USE statements involving the same module. As a result, it is possible for one use-associated entity to be accessible by more than one local identifier.

The local identifier of an entity made accessible by a USE statement shall not appear in any other nonexecutable statement that would cause any attribute (5.1.2) of the entity to be specified in the scoping unit that contains the USE statement, except that it may appear in a PUBLIC or PRIVATE statement in the scoping unit of a module and it may be given the ASYNCHRONOUS or VOLATILE attribute.

The appearance of such a local identifier in a PUBLIC statement in a module causes the entity accessible by the USE statement to be a public entity of that module. If the identifier appears in a PRIVATE statement in a module, the entity is not a public entity of that module. If the local identifier does not appear in either a PUBLIC or PRIVATE statement, it assumes the default accessibility attribute (5.2.1) of that scoping unit.

NOTE 11.10

The constraints in sections 5.5.1, 5.5.2, and 5.4 prohibit the *local-name* from appearing as a *common-block-object* in a COMMON statement, an *equivalence-object* in an EQUIVALENCE statement, or a *namelist-group-name* in a NAMELIST statement, respectively. There is no prohibition against the *local-name* appearing as a *common-block-name* or a *namelist-group-object*.

NOTE 11.11

For a discussion of the impact of the ONLY clause and renaming on dependent compilation, see section C.8.2.1.

NOTE 11.12

Examples:

```
USE STATS_LIB
```

provides access to all public entities in the module STATS_LIB.

```
USE MATH_LIB; USE STATS_LIB, SPROD => PROD
```

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.

```
USE STATS_LIB, ONLY: YPROD; USE STATS_LIB, ONLY : PROD
```

makes public entities YPROD and PROD in STATS_LIB accessible.

```
USE STATS_LIB, ONLY : YPROD; USE STATS_LIB
```

makes all public entities in STATS_LIB accessible.

11.3 Block data program units

A **block data program unit** is used to provide initial values for data objects in named common blocks.

R1116	*block-data*	**is**	*block-data-stmt*
			[*specification-part*]
			end-block-data-stmt
R1117	*block-data-stmt*	**is**	BLOCK DATA [*block-data-name*]
R1118	*end-block-data-stmt*	**is**	END [BLOCK DATA [*block-data-name*]]

C1116 (R1116) The *block-data-name* shall be included in the *end-block-data-stmt* only if it was provided in the *block-data-stmt* and, if included, shall be identical to the *block-data-name* in the *block-*

data-stmt.

C1117 (R1116) A *block-data specification-part* shall contain only derived-type definitions and ASYNCHRONOUS, BIND, COMMON, DATA, DIMENSION, EQUIVALENCE, IMPLICIT, INTRINSIC, PARAMETER, POINTER, SAVE, TARGET, USE, VOLATILE, and type declaration statements.

C1118 (R1116) A type declaration statement in a *block-data specification-part* shall not contain ALLOCATABLE, EXTERNAL, or BIND attribute specifiers.

NOTE 11.13

For explanatory information about the uses for the *block-data-name*, see section C.8.1.

If an object in a named common block is initially defined, all storage units in the common block storage sequence shall 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 program unit.

NOTE 11.14

In the example

```
     BLOCK DATA INIT
        REAL A, B, C, D, E, F
        COMMON /BLOCK1/ A, B, C, D
        DATA A /1.2/, C /2.3/
        COMMON /BLOCK2/ E, F
        DATA F /6.5/
     END BLOCK DATA INIT
```

common blocks BLOCK1 and BLOCK2 both have objects that are being initialized in a single block data program unit. B, D, and E are not initialized but they need to be specified as part of the common blocks.

Only an object in a named common block may be initially defined in a block data program unit.

NOTE 11.15

Objects associated with an object in a common block are considered to be in that common block.

The same named common block shall not be specified in more than one block data program unit in a program.

There shall not be more than one unnamed block data program unit in a program.

NOTE 11.16

An example of a block data program unit is:

```
BLOCK DATA WORK
   COMMON /WRKCOM/ A, B, C (10, 10)
   REAL :: A, B, C
   DATA A /1.0/, B /2.0/, C /100 * 0.0/
END BLOCK DATA WORK
```

Section 12: Procedures

The concept of a procedure was introduced in 2.2.3. This section contains a complete description of procedures. The actions specified by a procedure are performed when the procedure is invoked by execution of a reference to it.

The sequence of actions encapsulated by a procedure has access to entities in the invoking scoping unit by way of argument association (12.4.1). A **dummy argument** is a name that appears in the SUBROUTINE, FUNCTION, or ENTRY statement in the declaration of a procedure (R1233). Dummy arguments are also specified for intrinsic procedures and procedures in intrinsic modules in Sections 13, 14, and 15.

The entities in the invoking scoping unit are specified by actual arguments. An **actual argument** is an entity that appears in a procedure reference (R1221).

A procedure may also have access to entities in other scoping units, not necessarily the invoking scoping unit, by use association (16.4.1.2), host association (16.4.1.3), linkage association (16.4.1.4), storage association (16.4.3), or by reference to external procedures (5.1.2.6).

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 either appears explicitly as a primary within an expression, or is implied by a defined operation (7.1.3) within an expression. A reference to a subroutine is a CALL statement or a defined assignment statement (7.4.1.4).

A procedure is classified as **elemental** if it is a procedure that may be referenced elementally (12.7).

12.1.2 Procedure classification by means of definition

A procedure is either an intrinsic procedure, an external procedure, a module procedure, an internal procedure, a dummy procedure (which may be a dummy procedure pointer), a nondummy procedure pointer, 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, internal, and module procedures

An **external procedure** is a procedure that is defined by an external subprogram or by a means other than Fortran.

An **internal procedure** is a procedure that is defined by an internal subprogram. Internal subprograms may appear in the main program, in an external subprogram, or in a module subprogram. Internal subprograms shall not appear in other internal subprograms. Internal subprograms are the same as external subprograms except that the name of the internal procedure is not a global identifier, an internal subprogram shall not contain an ENTRY statement, the internal procedure name shall not be argument associated with a dummy procedure (12.4.1.3), and the internal subprogram has access to host entities by host association.

A **module procedure** is a procedure that is defined by a module subprogram.

A subprogram defines a procedure for the SUBROUTINE or FUNCTION statement. If the subprogram has one or more ENTRY statements, it also defines a procedure for each of them.

12.1.2.3 Dummy procedures

A dummy argument that is specified to be a procedure or appears in a procedure reference is a **dummy procedure**. A dummy procedure with the POINTER attribute is a dummy procedure pointer.

12.1.2.4 Procedure pointers

A procedure pointer is a procedure that has the EXTERNAL and POINTER attributes; it may be pointer associated with an external procedure, a module procedure, an intrinsic procedure, or a dummy procedure that is not a procedure pointer.

12.1.2.5 Statement functions

A function that is defined by a single statement is a **statement function** (12.5.4).

12.2 Characteristics of procedures

The **characteristics of a procedure** are the classification of the procedure as a function or subroutine, whether it is pure, whether it is elemental, whether it has the BIND attribute, the characteristics of its dummy arguments, and the characteristics of its result value if it is a function.

12.2.1 Characteristics of dummy arguments

Each dummy argument has the characteristic that it is a dummy data object, a dummy procedure, a dummy procedure pointer, or an asterisk (alternate return indicator). A dummy argument other than an asterisk 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 are its type, its type parameters (if any), its shape, its intent (5.1.2.7, 5.2.7), whether it is optional (5.1.2.9, 5.2.8), whether it is allocatable (5.1.2.5.3), whether it has the VALUE (5.1.2.15), ASYNCHRONOUS (5.1.2.3), or VOLATILE (5.1.2.16) attributes, whether it is polymorphic, and whether it is a pointer (5.1.2.11, 5.2.10) or a target (5.1.2.14, 5.2.13). If a type parameter of an object or a bound of an array is not an initialization expression, the exact dependence on the entities in the expression is a characteristic. If a shape, size, or type parameter is assumed or deferred, it is a characteristic.

12.2.1.2 Characteristics of dummy procedures and dummy procedure pointers

The characteristics of a dummy procedure are the explicitness of its interface (12.3.1), its characteristics as a procedure if the interface is explicit, whether it is a pointer, and whether it is optional (5.1.2.9, 5.2.8).

12.2.1.3 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 are its type, type parameters (if any), rank, whether it is polymorphic, whether it is allocatable, whether it is a pointer, and whether it is a procedure pointer. If a function result is an array that is not allocatable or a pointer, its shape is a characteristic. If a type parameter of a function result or a bound of a function result array is not an initialization expression, the exact dependence on the entities in the expression is a characteristic. If type parameters of a function result are deferred, which parameters are deferred is a characteristic. If the length of a character function result 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 procedure's interface consists of its abstract interface, its name, its binding label if any, and the procedure's generic identifiers, if any. The characteristics of a procedure are fixed, but the remainder of the interface may differ in different scoping units.

An **abstract interface** consists of procedure characteristics and the names of dummy arguments.

NOTE 12.1

For more explanatory information on procedure interfaces, see C.9.3.

12.3.1 Implicit and explicit interfaces

If a procedure is accessible in a scoping unit, its interface is either **explicit** or **implicit** in that scoping unit. The interface of an internal procedure, module procedure, or intrinsic procedure is always explicit in such a scoping unit. The interface of a subroutine or a function with a separate result name is explicit within the subprogram that defines it. The interface of a statement function is always implicit. The interface of an external procedure or dummy procedure is explicit in a scoping unit other than its own if an interface body (12.3.2.1) for the procedure is supplied or accessible, and implicit otherwise.

NOTE 12.2

For example, the subroutine LLS of C.8.3.5 has an explicit interface.

12.3.1.1 Explicit interface

A procedure other than a statement function shall have an explicit interface if it is referenced and

(1) A reference to the procedure appears

  (a) With an argument keyword (12.4.1),

  (b) As a reference by its generic name (12.3.2.1),

  (c) As a defined assignment (subroutines only),

  (d) In an expression as a defined operator (functions only), or

  (e) In a context that requires it to be pure,

(2) The procedure has a dummy argument that

  (a) has the ALLOCATABLE, ASYNCHRONOUS, OPTIONAL, POINTER, TARGET, VALUE, or VOLATILE attribute,

  (b) is an assumed-shape array,

  (c) is of a parameterized derived type, or

  (d) is polymorphic,

(3) The procedure has a result that
  (a) is an array,
  (b) is a pointer or is allocatable, or
  (c) has a nonassumed type parameter value that is not an initialization expression,

(4) The procedure is elemental, or

(5) The procedure has the BIND attribute.

12.3.2 Specification of the procedure interface

The interface for an internal, external, module, 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 body, or both, except that the ENTRY statement shall not appear in an interface body.

NOTE 12.3

An interface body cannot be used to describe the interface of an internal procedure, a module procedure, or an intrinsic procedure because the interfaces of such procedures are already explicit. However, the name of a procedure may appear in a PROCEDURE statement in an interface block (12.3.2.1).

12.3.2.1 Interface block

R1201	*interface-block*	**is**	*interface-stmt*
			[*interface-specification*] ...
			end-interface-stmt
R1202	*interface-specification*	**is**	*interface-body*
		or	*procedure-stmt*
R1203	*interface-stmt*	**is**	INTERFACE [*generic-spec*]
		or	ABSTRACT INTERFACE
R1204	*end-interface-stmt*	**is**	END INTERFACE [*generic-spec*]
R1205	*interface-body*	**is**	*function-stmt*
			[*specification-part*]
			end-function-stmt
		or	*subroutine-stmt*
			[*specification-part*]
			end-subroutine-stmt
R1206	*procedure-stmt*	**is**	[MODULE] PROCEDURE *procedure-name-list*
R1207	*generic-spec*	**is**	*generic-name*
		or	OPERATOR (*defined-operator*)
		or	ASSIGNMENT (=)
		or	*dtio-generic-spec*
R1208	*dtio-generic-spec*	**is**	READ (FORMATTED)
		or	READ (UNFORMATTED)
		or	WRITE (FORMATTED)
		or	WRITE (UNFORMATTED)
R1209	*import-stmt*	**is**	IMPORT [[::] *import-name-list*]

C1201 (R1201) An *interface-block* in a subprogram shall not contain an *interface-body* for a procedure defined by that subprogram.

C1202 (R1201) The *generic-spec* shall be included in the *end-interface-stmt* only if it is provided in the *interface-stmt*. If the *end-interface-stmt* includes *generic-name*, the *interface-stmt* shall specify the same *generic-name*. If the *end-interface-stmt* includes ASSIGNMENT(=), the *interface-*

stmt shall specify ASSIGNMENT(=). If the *end-interface-stmt* includes *dtio-generic-spec*, the *interface-stmt* shall specify the same *dtio-generic-spec*. If the *end-interface-stmt* includes OPERATOR(*defined-operator*), the *interface-stmt* shall specify the same *defined-operator*. If one *defined-operator* is .LT., .LE., .GT., .GE., .EQ., or .NE., the other is permitted to be the corresponding operator <, <=, >, >=, ==, or /=.

C1203 (R1203) If the *interface-stmt* is ABSTRACT INTERFACE, then the *function-name* in the *function-stmt* or the *subroutine-name* in the *subroutine-stmt* shall not be the same as a keyword that specifies an intrinsic type.

C1204 (R1202) A *procedure-stmt* is allowed only in an interface block that has a *generic-spec*.

C1205 (R1205) An *interface-body* of a pure procedure shall specify the intents of all dummy arguments except pointer, alternate return, and procedure arguments.

C1206 (R1205) An *interface-body* shall not contain an *entry-stmt*, *data-stmt*, *format-stmt*, or *stmt-function-stmt*.

C1207 (R1206) A *procedure-name* shall have an explicit interface and shall refer to an accessible procedure pointer, external procedure, dummy procedure, or module procedure.

C1208 (R1206) If MODULE appears in a *procedure-stmt*, each *procedure-name* in that statement shall be accessible in the current scope as a module procedure.

C1209 (R1206) A *procedure-name* shall not specify a procedure that is specified previously in any *procedure-stmt* in any accessible interface with the same generic identifier.

C1210 (R1209) The IMPORT statement is allowed only in an *interface-body*.

C1211 (R1209) Each *import-name* shall be the name of an entity in the host scoping unit.

An external or module subprogram specifies a **specific interface** for the procedures defined in that subprogram. Such a specific interface is explicit for module procedures and implicit for external procedures.

An interface block introduced by ABSTRACT INTERFACE is an **abstract interface block**. An interface body in an abstract interface block specifies an abstract interface. An interface block with a generic specification is a **generic interface block**. An interface block with neither ABSTRACT nor a generic specification is a **specific interface block**.

The name of the entity declared by an interface body is the *function-name* in the *function-stmt* or the *subroutine-name* in the *subroutine-stmt* that begins the interface body.

An interface body in a generic or specific interface block specifies the EXTERNAL attribute and an explicit specific interface for an external procedure or a dummy procedure. If the name of the declared procedure is that of a dummy argument in the subprogram containing the interface body, the procedure is a dummy procedure; otherwise, it is an external procedure.

An interface body specifies all of the characteristics of the explicit specific interface or abstract interface. The specification part of an interface body may specify attributes or define values for data entities that do not determine characteristics of the procedure. Such specifications have no effect.

If an explicit specific interface is specified by an interface body or a procedure declaration statement (12.3.2.3) for an external procedure, the characteristics shall be consistent with those specified in the procedure definition, except that the interface may specify a procedure that is not pure if the procedure is defined to be pure. An interface for a procedure named by an ENTRY statement may be specified by using the entry name as the procedure name in the interface body. An explicit specific interface may be specified by an interface body for an external procedure that does not exist in the program if the procedure is never used in any way. A procedure shall not have more than one explicit specific interface in a given scoping unit.

NOTE 12.4

The dummy argument names may be different because the name of a dummy argument is not a characteristic.

The IMPORT statement specifies that the named entities from the host scoping unit are accessible in the interface body by host association. An entity that is imported in this manner and is defined in the host scoping unit shall be explicitly declared prior to the interface body. The name of an entity made accessible by an IMPORT statement shall not appear in any of the contexts described in 16.4.1.3 that cause the host entity of that name to be inaccessible.

Within an interface body, if an IMPORT statement with no *import-name-list* appears, each host entity not named in an IMPORT statement also is made accessible by host association if its name does not appear in any of the contexts described in 16.4.1.3 that cause the host entity of that name to be inaccessible.

NOTE 12.5

An example of an interface block without a generic specification is:

```
INTERFACE
   SUBROUTINE EXT1 (X, Y, Z)
      REAL, DIMENSION (100, 100) :: X, Y, Z
   END SUBROUTINE EXT1
   SUBROUTINE EXT2 (X, Z)
      REAL X
      COMPLEX (KIND = 4) Z (2000)
   END SUBROUTINE EXT2
   FUNCTION EXT3 (P, Q)
      LOGICAL EXT3
      INTEGER P (1000)
      LOGICAL Q (1000)
   END FUNCTION EXT3
END INTERFACE
```

This interface block specifies explicit interfaces for the three external procedures EXT1, EXT2, and EXT3. Invocations of these procedures may use argument keywords (12.4.1); for example:

```
EXT3 (Q = P_MASK (N+1 : N+1000), P = ACTUAL_P)
```

NOTE 12.6

The IMPORT statement can be used to allow module procedures to have dummy arguments that are procedures with assumed-shape arguments of an opaque type. For example:

```
MODULE M
  TYPE T
    PRIVATE    ! T is an opaque type
    ...
  END TYPE
CONTAINS
  SUBROUTINE PROCESS(X,Y,RESULT,MONITOR)
    TYPE(T),INTENT(IN) :: X(:,:),Y(:,:)
```

NOTE 12.6 (cont.)

```
    TYPE(T),INTENT(OUT) :: RESULT(:,:)
    INTERFACE
      SUBROUTINE MONITOR(ITERATION_NUMBER,CURRENT_ESTIMATE)
        IMPORT T
        INTEGER,INTENT(IN) :: ITERATION_NUMBER
        TYPE(T),INTENT(IN) :: CURRENT_ESTIMATE(:,:)
      END SUBROUTINE
    END INTERFACE
    ...
  END SUBROUTINE
END MODULE
```

The MONITOR dummy procedure requires an explicit interface because it has an assumed-shape array argument, but TYPE(T) would not be available inside the interface body without the IMPORT statement.

A generic interface block specifies a **generic interface** for each of the procedures in the interface block. The PROCEDURE statement lists procedure pointers, external procedures, dummy procedures, or module procedures that have this generic interface. A generic interface is always explicit.

Any procedure may be referenced via its specific interface if the specific interface is accessible. It also may be referenced via a generic interface. The *generic-spec* in an *interface-stmt* is a **generic identifier** for all the procedures in the interface block. The rules specifying how any two procedures with the same generic identifier shall differ are given in 16.2.3. They ensure that any generic invocation applies to at most one specific procedure.

A **generic name** specifies a single name to reference all of the procedure names in the interface block. A generic name may be the same as any one of the procedure names in the interface block, or the same as any accessible generic name.

A generic name may be the same as a derived-type name, in which case all of the procedures in the interface block shall be functions.

NOTE 12.7

An example of a generic procedure interface is:

```
INTERFACE SWITCH
   SUBROUTINE INT_SWITCH (X, Y)
      INTEGER, INTENT (INOUT) :: X, Y
   END SUBROUTINE INT_SWITCH
   SUBROUTINE REAL_SWITCH (X, Y)
      REAL, INTENT (INOUT) :: X, Y
   END SUBROUTINE REAL_SWITCH
   SUBROUTINE COMPLEX_SWITCH (X, Y)
      COMPLEX, INTENT (INOUT) :: X, Y
   END SUBROUTINE COMPLEX_SWITCH
END INTERFACE SWITCH
```

Any of these three subroutines (INT_SWITCH, REAL_SWITCH, COMPLEX_SWITCH) may be referenced with the generic name SWITCH, as well as by its specific name. For example, a reference to INT_SWITCH could take the form:

NOTE 12.7 (cont.)

```
CALL SWITCH (MAX_VAL, LOC_VAL) ! MAX_VAL and LOC_VAL are of type INTEGER
```

An *interface-stmt* having a *dtio-generic-spec* is an interface for a user-defined derived-type input/output procedure (9.5.3.7)

12.3.2.1.1 Defined operations

If OPERATOR is specified in a generic specification, all of the procedures specified in the generic interface shall be functions that may be referenced as defined operations (7.1.3, 7.1.8.7, 7.2, 12.4). In the case of functions of two arguments, infix binary operator notation is implied. In the case of functions of one argument, prefix operator notation is implied. OPERATOR shall not be specified for functions with no arguments or for functions with more than two arguments. The dummy arguments shall be nonoptional dummy data objects and shall be specified with INTENT (IN). The function result shall not have assumed character length. If the operator is an *intrinsic-operator* (R310), the number of function arguments shall be consistent with the intrinsic uses of that operator, and the types, kind type parameters, or ranks of the dummy arguments shall differ from those required for the intrinsic operation (7.1.2).

A defined operation is treated as a reference to the function. For a unary defined operation, the operand corresponds to the function's dummy argument; for a binary operation, the left-hand operand corresponds to the first dummy argument of the function and the right-hand operand corresponds to the second dummy argument.

NOTE 12.8

An example of the use of the OPERATOR generic specification is:

```
INTERFACE OPERATOR ( * )
   FUNCTION BOOLEAN_AND (B1, B2)
      LOGICAL, INTENT (IN) :: B1 (:), B2 (SIZE (B1))
      LOGICAL :: BOOLEAN_AND (SIZE (B1))
   END FUNCTION BOOLEAN_AND
END INTERFACE OPERATOR ( * )
```

This allows, for example

```
SENSOR (1:N) * ACTION (1:N)
```

as an alternative to the function call

```
BOOLEAN_AND (SENSOR (1:N), ACTION (1:N))   ! SENSOR and ACTION are
                                           ! of type LOGICAL
```

A given defined operator may, as with generic names, apply to more than one function, in which case it is generic in exact analogy to generic procedure names. For intrinsic operator symbols, the generic properties include the intrinsic operations they represent. Because both forms of each relational operator have the same interpretation (7.2), extending one form (such as <=) has the effect of defining both forms (<= and .LE.).

NOTE 12.9

In Fortran 90 and Fortran 95, it was not possible to define operators on pointers because pointer dummy arguments were disallowed from having an INTENT attribute. The restriction against INTENT for pointer dummy arguments is now lifted, so defined operations on pointers are now

NOTE 12.9 (cont.)

possible.

However, the POINTER attribute cannot be used to resolve generic procedures (16.2.3), so it is not possible to define a generic operator that has one procedure for pointers and another procedure for nonpointers.

12.3.2.1.2 Defined assignments

If ASSIGNMENT (=) is specified in a generic specification, all the procedures in the generic interface shall be subroutines that may be referenced as defined assignments (7.4.1.4). Defined assignment may, as with generic names, apply to more than one subroutine, in which case it is generic in exact analogy to generic procedure names.

Each of these subroutines shall have exactly two dummy arguments. Each argument shall be nonoptional. The first argument shall have INTENT (OUT) or INTENT (INOUT) and the second argument shall have INTENT (IN). Either the second argument shall be an array whose rank differs from that of the first argument, the declared types and kind type parameters of the arguments shall not conform as specified in Table 7.8, or the first argument shall be of derived type. A defined assignment is treated as a reference to the subroutine, with the left-hand side as the first argument and the right-hand side enclosed in parentheses as the second argument. The ASSIGNMENT generic specification specifies that assignment is extended or redefined.

NOTE 12.10

An example of the use of the ASSIGNMENT generic specification is:

```
INTERFACE ASSIGNMENT ( = )

   SUBROUTINE LOGICAL_TO_NUMERIC (N, B)
      INTEGER, INTENT (OUT) :: N
      LOGICAL, INTENT (IN)  :: B
   END SUBROUTINE LOGICAL_TO_NUMERIC
   SUBROUTINE CHAR_TO_STRING (S, C)
      USE STRING_MODULE       ! Contains definition of type STRING
      TYPE (STRING), INTENT (OUT) :: S  ! A variable-length string
      CHARACTER (*), INTENT (IN) ::  C
   END SUBROUTINE CHAR_TO_STRING
END INTERFACE ASSIGNMENT ( = )
```

Example assignments are:

```
KOUNT = SENSOR (J)   ! CALL LOGICAL_TO_NUMERIC (KOUNT, (SENSOR (J)))
NOTE  = '89AB'       ! CALL CHAR_TO_STRING (NOTE, ('89AB'))
```

12.3.2.1.3 User-defined derived-type input/output procedure interfaces

All of the procedures specified in an interface block for a user-defined derived-type input/output procedure shall be subroutines that have interfaces as described in 9.5.3.7.2.

12.3.2.2 EXTERNAL statement

An **EXTERNAL statement** specifies the EXTERNAL attribute (5.1.2.6) for a list of names.

R1210 *external-stmt* **is** EXTERNAL [::] *external-name-list*

Each *external-name* shall be the name of an external procedure, a dummy argument, a procedure pointer, or a block data program unit. In an external subprogram, an EXTERNAL statement shall not specify the name of a procedure defined by the subprogram.

The appearance of the name of a block data program unit in an EXTERNAL statement confirms that the block data program unit is a part of the program.

NOTE 12.11

For explanatory information on potential portability problems with external procedures, see section C.9.1.

NOTE 12.12

An example of an EXTERNAL statement is:

```
EXTERNAL FOCUS
```

12.3.2.3 Procedure declaration statement

A procedure declaration statement declares procedure pointers, dummy procedures, and external procedures. It specifies the EXTERNAL attribute (5.1.2.6) for all procedure entities in the *proc-decl-list*.

R1211	*procedure-declaration-stmt*	**is**	PROCEDURE ([*proc-interface*]) ■
			■ [[, *proc-attr-spec*] ... ::] *proc-decl-list*
R1212	*proc-interface*	**is**	*interface-name*
		or	*declaration-type-spec*
R1213	*proc-attr-spec*	**is**	*access-spec*
		or	*proc-language-binding-spec*
		or	INTENT (*intent-spec*)
		or	OPTIONAL
		or	POINTER
		or	SAVE
R1214	*proc-decl*	**is**	*procedure-entity-name* [=> *null-init*]
R1215	*interface-name*	**is**	*name*

C1212 (R1215) The *name* shall be the name of an abstract interface or of a procedure that has an explicit interface. If *name* is declared by a *procedure-declaration-stmt* it shall be previously declared. If *name* denotes an intrinsic procedure it shall be one that is listed in 13.6 and not marked with a bullet (•).

C1213 (R1215) The *name* shall not be the same as a keyword that specifies an intrinsic type.

C1214 If a procedure entity has the INTENT attribute or SAVE attribute, it shall also have the POINTER attribute.

C1215 (R1211) If a *proc-interface* describes an elemental procedure, each *procedure-entity-name* shall specify an external procedure.

C1216 (R1214) If => appears in *proc-decl*, the procedure entity shall have the POINTER attribute.

C1217 (R1211) If *proc-language-binding-spec* with a NAME= is specified, then *proc-decl-list* shall contain exactly one *proc-decl*, which shall neither have the POINTER attribute nor be a dummy procedure.

C1218 (R1211) If *proc-language-binding-spec* is specified, the *proc-interface* shall appear, it shall be an *interface-name*, and *interface-name* shall be declared with a *proc-language-binding-spec*.

If *proc-interface* appears and consists of *interface-name*, it specifies an explicit specific interface (12.3.2.1) for the declared procedures or procedure pointers. The abstract interface (12.3) is that specified by the interface named by *interface-name*.

If *proc-interface* appears and consists of *declaration-type-spec*, it specifies that the declared procedures or procedure pointers are functions having implicit interfaces and the specified result type. If a type is specified for an external function, its function definition (12.5.2.1) shall specify the same result type and type parameters.

If *proc-interface* does not appear, the procedure declaration statement does not specify whether the declared procedures or procedure pointers are subroutines or functions.

If a *proc-attr-spec* other than a *proc-language-binding-spec* appears, it specifies that the declared procedures or procedure pointers have that attribute. These attributes are described in 5.1.2. If a *proc-language-binding-spec* with NAME= appears, it specifies a binding label as described in 15.4.1. A *proc-language-binding-spec* without a NAME= is allowed, but is redundant with the *proc-interface* required by C1218.

If => *null-init* appears in a *proc-decl* in a *procedure-declaration-stmt*, it specifies that the initial association status of the corresponding procedure entity is disassociated. It also implies the SAVE attribute, which may be reaffirmed by explicit use of the SAVE attribute in the *procedure-declaration-stmt* or by a SAVE statement."

NOTE 12.13

In contrast to the EXTERNAL statement, it is not possible to use the PROCEDURE statement to identify a BLOCK DATA subprogram.

NOTE 12.14

The following code illustrates PROCEDURE statements. Note 7.43 illustrates the use of the P and BESSEL defined by this code.

```
ABSTRACT INTERFACE
  FUNCTION REAL_FUNC (X)
    REAL, INTENT (IN) :: X
    REAL :: REAL_FUNC
  END FUNCTION REAL_FUNC
END INTERFACE

INTERFACE
  SUBROUTINE SUB (X)
    REAL, INTENT (IN) :: X
  END SUBROUTINE SUB
END INTERFACE

!-- Some external or dummy procedures with explicit interface.
PROCEDURE (REAL_FUNC) :: BESSEL, GAMMA
PROCEDURE (SUB) :: PRINT_REAL
!-- Some procedure pointers with explicit interface,
!-- one initialized to NULL().
PROCEDURE (REAL_FUNC), POINTER :: P, R => NULL()
PROCEDURE (REAL_FUNC), POINTER :: PTR_TO_GAMMA
!-- A derived type with a procedure pointer component ...
TYPE STRUCT_TYPE
   PROCEDURE (REAL_FUNC), POINTER :: COMPONENT
```

NOTE 12.14 (cont.)

```
END TYPE STRUCT_TYPE
!-- ... and a variable of that type.
TYPE(STRUCT_TYPE) :: STRUCT
!-- An external or dummy function with implicit interface
PROCEDURE (REAL) :: PSI
```

12.3.2.4 INTRINSIC statement

An **INTRINSIC statement** specifies a list of names that have the INTRINSIC attribute (5.1.2.8).

R1216 *intrinsic-stmt* **is** INTRINSIC [::] *intrinsic-procedure-name-list*

C1219 (R1216) Each *intrinsic-procedure-name* shall be the name of an intrinsic procedure.

NOTE 12.15

A name shall not be explicitly specified to have both the EXTERNAL and INTRINSIC attributes in the same scoping unit.

12.3.2.5 Implicit interface specification

In a scoping unit where the interface of a function is implicit, the type and type parameters of the function result are specified by an implicit or explicit type specification of the function name. The type, type parameters, and shape of dummy arguments of a procedure referenced from a scoping unit where the interface of the procedure is implicit shall 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 or procedure pointer, but is independent of the means by which the procedure is defined. The forms of procedure references are:

R1217 *function-reference* **is** *procedure-designator* ([*actual-arg-spec-list*])

C1220 (R1217) The *procedure-designator* shall designate a function.

C1221 (R1217) The *actual-arg-spec-list* shall not contain an *alt-return-spec*.

R1218 *call-stmt* **is** CALL *procedure-designator* [([*actual-arg-spec-list*])]

C1222 (R1218) The *procedure-designator* shall designate a subroutine.

R1219 *procedure-designator* **is** *procedure-name*
or *proc-component-ref*
or *data-ref* % *binding-name*

C1223 (R1219) A *procedure-name* shall be the name of a procedure or procedure pointer.

C1224 (R1219) A *binding-name* shall be a binding name (4.5.4) of the declared type of *data-ref*.

Resolving references to type-bound procedures is described in 12.4.5.

A function may also be referenced as a defined operation (12.3.2.1.1). A subroutine may also be referenced as a defined assignment (12.3.2.1.2).

R1220 *actual-arg-spec* **is** [*keyword* =] *actual-arg*
R1221 *actual-arg* **is** *expr*
or *variable*
or *procedure-name*
or *proc-component-ref*
or *alt-return-spec*
R1222 *alt-return-spec* **is** * *label*

C1225 (R1220) The *keyword* = shall not appear if the interface of the procedure is implicit in the scoping unit.

C1226 (R1220) The *keyword* = shall not be omitted from an *actual-arg-spec* unless it has been omitted from each preceding *actual-arg-spec* in the argument list.

C1227 (R1220) Each *keyword* shall be the name of a dummy argument in the explicit interface of the procedure.

C1228 (R1221) A nonintrinsic elemental procedure shall not be used as an actual argument.

C1229 (R1221) A *procedure-name* shall be the name of an external procedure, a dummy procedure, a module procedure, a procedure pointer, or a specific intrinsic function that is listed in 13.6 and not marked with a bullet(•).

NOTE 12.16

This standard does not allow internal procedures to be used as actual arguments, in part to simplify the problem of ensuring that internal procedures with recursive hosts access entities from the correct instance (12.5.2.3) of the host. If, as an extension, a processor allows internal procedures to be used as actual arguments, the correct instance in this case is the instance in which the procedure is supplied as an actual argument, even if the corresponding dummy argument is eventually invoked from a different instance.

C1230 (R1221) In a reference to a pure procedure, a *procedure-name actual-arg* shall be the name of a pure procedure (12.6).

NOTE 12.17

This constraint ensures that the purity of a procedure cannot be undermined by allowing it to call a nonpure procedure.

C1231 (R1222) The *label* used in the *alt-return-spec* shall be the statement label of a branch target statement that appears in the same scoping unit as the *call-stmt*.

NOTE 12.18

Successive commas shall not be used to omit optional arguments.

NOTE 12.19

Examples of procedure reference using procedure pointers:

```
P => BESSEL
WRITE (*, *) P(2.5)     !-- BESSEL(2.5)

S => PRINT_REAL
CALL S(3.14)
```

12.4.1 Actual arguments, dummy arguments, and argument association

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. This correspondence may be established either by keyword or by position. If an argument keyword appears, the actual argument is associated with the dummy argument whose name is the same as the argument keyword (using the dummy argument names from the interface accessible in the scoping unit containing the procedure reference). In the absence of an argument keyword, an actual argument is associated with the dummy argument occupying the corresponding position in the reduced dummy argument list; that is, the first actual argument is associated with the first dummy argument in the reduced list, the second actual argument is associated with the second dummy argument in the reduced list, etc. The reduced dummy argument list is either the full dummy argument list or, if there is a passed-object dummy argument (4.5.3.3), the dummy argument list with the passed object dummy argument omitted. Exactly one actual argument shall be associated with each nonoptional dummy argument. At most one actual argument may be associated with each optional dummy argument. Each actual argument shall be associated with a dummy argument.

NOTE 12.20

For example, the procedure defined by

```
SUBROUTINE SOLVE (FUNCT, SOLUTION, METHOD, STRATEGY, PRINT)
   INTERFACE
      FUNCTION FUNCT (X)
         REAL FUNCT, X
      END FUNCTION FUNCT
   END INTERFACE
   REAL SOLUTION
   INTEGER, OPTIONAL :: METHOD, STRATEGY, PRINT
   ...
```

may be invoked with

```
CALL SOLVE (FUN, SOL, PRINT = 6)
```

provided its interface is explicit; if the interface is specified by an interface block, the name of the last argument shall be PRINT.

12.4.1.1 The passed-object dummy argument and argument association

In a reference to a type-bound procedure that has a passed-object dummy argument (4.5.3.3), the *data-ref* of the *function-reference* or *call-stmt* is associated, as an actual argument, with the passed-object dummy argument.

12.4.1.2 Actual arguments associated with dummy data objects

If a dummy argument is neither allocatable nor a pointer, it shall be type compatible (5.1.1.2) with the associated actual argument. If a dummy argument is allocatable or a pointer, the associated actual argument shall be polymorphic if and only if the dummy argument is polymorphic, and the declared type of the actual argument shall be the same as the declared type of the dummy argument.

NOTE 12.21

The dynamic type of a polymorphic allocatable or pointer dummy argument may change as a result of execution of an allocate statement or pointer assignment in the subprogram. Because of this the corresponding actual argument needs to be polymorphic and have a declared type that

NOTE 12.21 (cont.)

> is the same as the declared type of the dummy argument or an extension of that type. However, type compatibility requires that the declared type of the dummy argument be the same as, or an extension of, the type of the actual argument. Therefore, the dummy and actual arguments need to have the same declared type.
>
> Dynamic type information is not maintained for a non-polymorphic allocatable or pointer dummy argument. However, allocating or pointer assigning such a dummy argument would require maintenance of this information if the corresponding actual argument is polymorphic. Therefore, the corresponding actual argument needs to be non-polymorphic.

The type parameter values of the actual argument shall agree with the corresponding ones of the dummy argument that are not assumed or deferred, except for the case of the character length parameter of an actual argument of type default character associated with a dummy argument that is not assumed shape.

If a scalar dummy argument is of type default character, the length *len* of the dummy argument shall be less than or equal to the length of the actual argument. The dummy argument becomes associated with the leftmost *len* characters of the actual argument. If an array dummy argument is of type default character and is not assumed shape, it becomes associated with the leftmost characters of the actual argument element sequence (12.4.1.5) and it shall not extend beyond the end of that sequence.

The values of assumed type parameters of a dummy argument are assumed from the corresponding type parameters of the associated actual argument.

An actual argument associated with a dummy argument that is allocatable or a pointer shall have deferred the same type parameters as the dummy argument.

If the dummy argument is a pointer, the actual argument shall be a pointer and the nondeferred type parameters and ranks shall agree. If a dummy argument is allocatable, the actual argument shall be allocatable and the nondeferred type parameters and ranks shall agree. It is permissible for the actual argument to have an allocation status of unallocated.

At the invocation of the procedure, the pointer association status of an actual argument associated with a pointer dummy argument becomes undefined if the dummy argument has INTENT(OUT).

Except in references to intrinsic inquiry functions, if the dummy argument is not a pointer and the corresponding actual argument is a pointer, the actual argument shall be associated with a target and the dummy argument becomes argument associated with that target.

Except in references to intrinsic inquiry functions, if the dummy argument is not allocatable and the actual argument is allocatable, the actual argument shall be allocated.

If the dummy argument has the VALUE attribute it becomes associated with a definable anonymous data object whose initial value is that of the actual argument. Subsequent changes to the value or definition status of the dummy argument do not affect the actual argument.

NOTE 12.22

> Fortran argument association is usually similar to call by reference and call by value-result. If the VALUE attribute is specified, the effect is as if the actual argument is assigned to a temporary, and the temporary is then argument associated with the dummy argument. The actual mechanism by which this happens is determined by the processor.

If the dummy argument does not have the TARGET or POINTER attribute, any pointers associated with the actual argument do not become associated with the corresponding dummy argument on invocation of the procedure. If such a dummy argument is associated with an actual argument that is a dummy argument with the TARGET attribute, whether any pointers associated with the original actual argument become associated with the dummy argument with the TARGET attribute is processor dependent.

If the dummy argument has the TARGET attribute, does not have the VALUE attribute, and is either a scalar or an assumed-shape array, and the corresponding actual argument has the TARGET attribute but is not an array section with a vector subscript then

(1) Any pointers associated with the actual argument become associated with the corresponding dummy argument on invocation of the procedure and

(2) When execution of the procedure completes, any pointers that do not become undefined (16.4.2.1.3) and are associated with the dummy argument remain associated with the actual argument.

If the dummy argument has the TARGET attribute and is an explicit-shape array or is an assumed-size array, and the corresponding actual argument has the TARGET attribute but is not an array section with a vector subscript then

(1) On invocation of the procedure, whether any pointers associated with the actual argument become associated with the corresponding dummy argument is processor dependent and

(2) When execution of the procedure completes, the pointer association status of any pointer that is pointer associated with the dummy argument is processor dependent.

If the dummy argument has the TARGET attribute and the corresponding actual argument does not have the TARGET attribute or is an array section with a vector subscript, any pointers associated with the dummy argument become undefined when execution of the procedure completes.

If the dummy argument has the TARGET attribute and the VALUE attribute, any pointers associated with the dummy argument become undefined when execution of the procedure completes.

If the actual argument is scalar, the corresponding dummy argument shall be scalar unless the actual argument is of type default character, of type character with the C character kind (15.1), or is an element or substring of an element of an array that is not an assumed-shape or pointer array. If the procedure is nonelemental and is referenced by a generic name or as a defined operator or defined assignment, the ranks of the actual arguments and corresponding dummy arguments shall agree.

If a dummy argument is an assumed-shape array, the rank of the actual argument shall be the same as the rank of the dummy argument; the actual argument shall not be an assumed-size array (including an array element designator or an array element substring designator).

Except when a procedure reference is elemental (12.7), each element of an array 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 array element order (6.2.2.2).

NOTE 12.23

For type default character sequence associations, the interpretation of element is provided in 12.4.1.5.

A scalar dummy argument of a nonelemental procedure may be associated only with a scalar actual argument.

If a nonpointer dummy argument has INTENT (OUT) or INTENT (INOUT), the actual argument shall be definable. If a dummy argument has INTENT (OUT), the corresponding actual argument becomes undefined at the time the association is established, except for components of an object of derived type for which default initialization has been specified. If the dummy argument is not polymorphic and the type of the actual argument is an extension type of the type of the dummy argument, only the part of the actual argument that is of the same type as the dummy argument becomes undefined.

If the actual argument is an array section having a vector subscript, the dummy argument is not definable and shall not have the INTENT (OUT), INTENT (INOUT), VOLATILE, or ASYNCHRONOUS attributes.

NOTE 12.24

Argument intent specifications serve several purposes. See Note 5.14.

NOTE 12.25

For more explanatory information on argument association and evaluation, see section C.9.5. For more explanatory information on pointers and targets as dummy arguments, see section C.9.6.

C1232 (R1221) If an actual argument is an array section or an assumed-shape array, and the corresponding dummy argument has either the VOLATILE or ASYNCHRONOUS attribute, that dummy argument shall be an assumed-shape array.

C1233 (R1221) If an actual argument is a pointer array, and the corresponding dummy argument has either the VOLATILE or ASYNCHRONOUS attribute, that dummy argument shall be an assumed-shape array or a pointer array.

NOTE 12.26

The constraints on actual arguments that correspond to a dummy argument with either the ASYNCHRONOUS or VOLATILE attribute are designed to avoid forcing a processor to use the so-called copy-in/copy-out argument passing mechanism. Making a copy of actual arguments whose values are likely to change due to an asynchronous I/O operation completing or in some unpredictable manner will cause those new values to be lost when a called procedure returns and the copy-out overwrites the actual argument.

12.4.1.3 Actual arguments associated with dummy procedure entities

If a dummy argument is a procedure pointer, the associated actual argument shall be a procedure pointer, a reference to a function that returns a procedure pointer, or a reference to the NULL intrinsic function.

If a dummy argument is a dummy procedure without the POINTER attribute, the associated actual argument shall be the specific name of an external, module, dummy, or intrinsic procedure, an associated procedure pointer, or a reference to a function that returns an associated procedure pointer. The only intrinsic procedures permitted are those listed in 13.6 and not marked with a bullet (•). If the specific name is also a generic name, only the specific procedure is associated with the dummy argument.

If an external procedure name or a dummy procedure name is used as an actual argument, its interface shall be explicit or it shall be explicitly declared to have the EXTERNAL attribute.

If the interface of the dummy argument is explicit, the characteristics listed in 12.2 shall be the same for the associated actual argument and the corresponding dummy argument, except that a pure actual argument may be associated with a dummy argument that is not pure and an elemental intrinsic actual procedure may be associated with a dummy procedure (which is prohibited from being elemental).

If the interface of the dummy argument is implicit and either the name of the dummy argument is explicitly typed or it is referenced as a function, the dummy argument shall not be referenced as a subroutine and the actual argument shall be a function, function procedure pointer, or dummy procedure.

If the interface of the dummy argument is implicit and a reference to it appears as a subroutine reference, the actual argument shall be a subroutine, subroutine procedure pointer, or dummy procedure.

12.4.1.4 Actual arguments associated with alternate return indicators

If a dummy argument is an asterisk (12.5.2.2), the associated actual argument shall be an alternate return specifier (12.4).

12.4.1.5 Sequence association

An actual argument represents an **element sequence** if it is an array expression, an array element designator, a scalar of type default character, or a scalar of type character with the C character kind (15.1.1). If the actual argument is an array expression, the element sequence consists of the elements in array element order. If the actual argument is an array element designator, the element sequence consists of that array element and each element that follows it in array element order.

If the actual argument is of type default character or of type character with the C character kind, and is an array expression, array element, or array element substring designator, the element sequence consists of the storage units beginning with the first storage unit of the actual argument and continuing to the end of the array. The storage units of an array element substring designator are viewed as array elements consisting of consecutive groups of storage units having the character length of the dummy array.

If the actual argument is of type default character or of type character with the C character kind, and is a scalar that is not an array element or array element substring designator, the element sequence consists of the storage units of the actual argument.

NOTE 12.27

Some of the elements in the element sequence may consist of storage units from different elements of the original array.

An actual argument that represents an element sequence and corresponds to a dummy argument that is an array is sequence associated with the dummy argument if the dummy argument is an explicit-shape or assumed-size array. 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 shall 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.

12.4.1.6 Restrictions on dummy arguments not present

A dummy argument or an entity that is host associated with a dummy argument is not **present** if the dummy argument

(1) is not associated with an actual argument, or

(2) is associated with an actual argument that is not present.

Otherwise, it is present. A dummy argument that is not optional shall be present. An optional dummy argument that is not present is subject to the following restrictions:

(1) If it is a data object, it shall not be referenced or be defined. If it is of a type for which default initialization is specified for some component, the initialization has no effect.

(2) It shall not be used as the *data-target* or *proc-target* of a pointer assignment.

(3) If it is a procedure or procedure pointer, it shall not be invoked.

(4) It shall not be supplied as an actual argument corresponding to a nonoptional dummy argument other than as the argument of the PRESENT intrinsic function or as an argument of a function reference that meets the requirements of (6) or (8) in 7.1.7.

(5) A designator with it as the base object and with at least one component selector, array section selector, array element selector, or substring selector shall not be supplied as an actual argument.

(6) If it is an array, it shall not be supplied as an actual argument to an elemental procedure unless an array of the same rank is supplied as an actual argument corresponding to a nonoptional dummy argument of that elemental procedure.

(7) If it is a pointer, it shall not be allocated, deallocated, nullified, pointer-assigned, or supplied as an actual argument corresponding to an optional nonpointer dummy argument.

(8) If it is allocatable, it shall not be allocated, deallocated, or supplied as an actual argument corresponding to an optional nonallocatable dummy argument.

(9) If it has length type parameters, they shall not be the subject of an inquiry.

(10) It shall not be used as the *selector* in a SELECT TYPE or ASSOCIATE construct.

Except as noted in the list above, it may be supplied as an actual argument corresponding to an optional dummy argument, which is then also considered not to be associated with an actual argument.

12.4.1.7 Restrictions on entities associated with dummy arguments

While an entity is associated with a dummy argument, the following restrictions hold:

(1) Action that affects the allocation status of the entity or a subobject thereof shall be taken through the dummy argument. Action that affects the value of the entity or any subobject of it shall be taken only through the dummy argument unless

  (a) the dummy argument has the POINTER attribute or

  (b) the dummy argument has the TARGET attribute, the dummy argument does not have INTENT (IN), the dummy argument is a scalar object or an assumed-shape array, and the actual argument is a target other than an array section with a vector subscript.

NOTE 12.28

In

```
SUBROUTINE OUTER
   REAL, POINTER :: A (:)
   ...
   ALLOCATE (A (1:N))
   ...
   CALL INNER (A)
   ...
CONTAINS
   SUBROUTINE INNER (B)
      REAL :: B (:)
      ...
   END SUBROUTINE INNER
   SUBROUTINE SET (C, D)
      REAL, INTENT (OUT) :: C
      REAL, INTENT (IN) :: D
      C = D
   END SUBROUTINE SET
END SUBROUTINE OUTER
```

an assignment statement such as

NOTE 12.28 (cont.)

```
A (1) = 1.0
```

would not be permitted during the execution of INNER because this would be changing A without using B, but statements such as

```
B (1) = 1.0
```

or

```
CALL SET (B (1), 1.0)
```

would be allowed. Similarly,

```
DEALLOCATE (A)
```

would not be allowed because this affects the allocation of B without using B. In this case,

```
DEALLOCATE (B)
```

also would not be permitted. If B were declared with the POINTER attribute, either of the statements

```
DEALLOCATE (A)
```

and

```
DEALLOCATE (B)
```

would be permitted, but not both.

NOTE 12.29

If there is a partial or complete overlap between the actual arguments associated with two different dummy arguments of the same procedure and the dummy arguments have neither the POINTER nor TARGET attribute, the overlapped portions shall not be defined, redefined, or become undefined during the execution of the procedure. For example, in

```
CALL SUB (A (1:5), A (3:9))
```

A (3:5) shall not be defined, redefined, or become undefined through the first dummy argument because it is part of the argument associated with the second dummy argument and shall not be defined, redefined, or become undefined through the second dummy argument because it is part of the argument associated with the first dummy argument. A (1:2) remains definable through the first dummy argument and A (6:9) remains definable through the second dummy argument.

NOTE 12.30

This restriction applies equally to pointer targets. In

```
REAL, DIMENSION (10), TARGET :: A
REAL, DIMENSION (:), POINTER :: B, C
B => A (1:5)
C => A (3:9)
```

NOTE 12.30 (cont.)

```
CALL SUB (B, C)  ! The dummy arguments of SUB are neither pointers nor targets.
```

B (3:5) cannot be defined because it is part of the argument associated with the second dummy argument. C (1:3) cannot be defined because it is part of the argument associated with the first dummy argument. A (1:2) [which is B (1:2)] remains definable through the first dummy argument and A (6:9) [which is C (4:7)] remains definable through the second dummy argument.

NOTE 12.31

Because a nonpointer dummy argument declared with INTENT(IN) shall not be used to change the associated actual argument, the associated actual argument remains constant throughout the execution of the procedure.

(2) If the allocation status of the entity or a subobject thereof is affected through the dummy argument, then at any time during the execution of the procedure, either before or after the allocation or deallocation, it may be referenced only through the dummy argument. If the value the entity or any subobject of it is affected through the dummy argument, then at any time during the execution of the procedure, either before or after the definition, it may be referenced only through that dummy argument unless

(a) the dummy argument has the POINTER attribute or

(b) the dummy argument has the TARGET attribute, the dummy argument does not have INTENT (IN), the dummy argument is a scalar object or an assumed-shape array, and the actual argument is a target other than an array section with a vector subscript.

NOTE 12.32

In

```
MODULE DATA
   REAL :: W, X, Y, Z
END MODULE DATA

PROGRAM MAIN
   USE DATA
      ...
   CALL INIT (X)
      ...
END PROGRAM MAIN
SUBROUTINE INIT (V)
   USE DATA
      ...
   READ (*, *) V
      ...
END SUBROUTINE INIT
```

variable X shall not be directly referenced at any time during the execution of INIT because it is being defined through the dummy argument V. X may be (indirectly) referenced through V. W, Y, and Z may be directly referenced. X may, of course, be directly referenced once execution of INIT is complete.

NOTE 12.33

The restrictions on entities associated with dummy arguments are intended to facilitate a variety of optimizations in the translation of the subprogram, including implementations of argument association in which the value of an actual argument that is neither a pointer nor a target is maintained in a register or in local storage.

12.4.2 Function reference

A function is invoked during expression evaluation by a *function-reference* or by a defined operation (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. The characteristics of the function result (12.2.2) are determined by the interface of the function. A reference to an elemental function (12.7) is an elemental reference if one or more actual arguments are arrays and all array arguments have the same shape.

12.4.3 Subroutine reference

A subroutine is invoked by execution of a CALL statement, execution of a defined assignment statement (7.4.1.4), user-defined derived-type input/output(9.5.3.7.1), or finalization(4.5.5). When a subroutine is invoked, all actual argument expressions are evaluated, then the arguments are associated, and then the subroutine is executed. When the actions specified by the subroutine are completed, the execution of the CALL statement, the execution of the defined assignment statement, or the processing of an input or output list item 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. A reference to an elemental subroutine (12.7) is an elemental reference if there is at least one actual argument corresponding to an INTENT(OUT) or INTENT(INOUT) dummy argument, all such actual arguments are arrays, and all actual arguments are conformable.

12.4.4 Resolving named procedure references

The rules for interpreting a procedure reference depend on whether the procedure name in the reference is established by the available declarations and specifications to be generic in the scoping unit containing the reference, is established to be only specific in the scoping unit containing the reference, or is not established.

(1) A procedure name is established to be generic in a scoping unit

   (a) if that scoping unit contains an interface block with that name;

   (b) if that scoping unit contains an INTRINSIC attribute specification for that name and it is the name of a generic intrinsic procedure;

   (c) if that scoping unit contains a USE statement that makes that procedure name accessible and the corresponding name in the module is established to be generic; or

   (d) if that scoping unit contains no declarations of that name, that scoping unit has a host scoping unit, and that name is established to be generic in the host scoping unit.

(2) A procedure name is established to be only specific in a scoping unit if it is established to be specific and not established to be generic. It is established to be specific

   (a) if that scoping unit contains a module subprogram, internal subprogram, or statement function that defines a procedure with that name;

   (b) if that scoping unit contains an INTRINSIC attribute specification for that name and if it is the name of a specific intrinsic procedure;

   (c) if that scoping unit contains an explicit EXTERNAL attribute specification (5.1.2.6) for that name;

(d) if that scoping unit contains a USE statement that makes that procedure name accessible and the corresponding name in the module is established to be specific; or

(e) if that scoping unit contains no declarations of that name, that scoping unit has a host scoping unit, and that name is established to be specific in the host scoping unit.

(3) A procedure name is not established in a scoping unit if it is neither established to be generic nor established to be specific.

12.4.4.1 Resolving procedure references to names established to be generic

(1) If the reference is consistent with a nonelemental reference to one of the specific interfaces of a generic interface that has that name and either is in the scoping unit in which the reference appears or is made accessible by a USE statement in the scoping unit, the reference is to the specific procedure in the interface block that provides that interface. The rules in 16.2.3 ensure that there can be at most one such specific procedure.

(2) If (1) does not apply, if the reference is consistent with an elemental reference to one of the specific interfaces of a generic interface that has that name and either is in the scoping unit in which the reference appears or is made accessible by a USE statement in the scoping unit, the reference is to the specific elemental procedure in the interface block that provides that interface. The rules in 16.2.3 ensure that there can be at most one such specific elemental procedure.

NOTE 12.34

These rules allow particular instances of a generic function to be used for particular array ranks and a general elemental version to be used for other ranks. Given an interface block such as:

```
INTERFACE RANF
     ELEMENTAL FUNCTION SCALAR_RANF(X)
     REAL X
     END FUNCTION SCALAR_RANF
     FUNCTION VECTOR_RANDOM(X)
     REAL X(:)
     REAL VECTOR_RANDOM(SIZE(X))
     END FUNCTION VECTOR_RANDOM
END INTERFACE RANF
```

and a declaration such as:

```
REAL A(10,10), AA(10,10)
```

then the statement

```
A = RANF(AA)
```

is an elemental reference to SCALAR_RANF. The statement

```
A = RANF(AA(6:10,2))
```

is a nonelemental reference to VECTOR_RANDOM.

(3) If (1) and (2) do not apply, if the scoping unit contains either an INTRINSIC attribute specification for that name or a USE statement that makes that name accessible from a module in which the corresponding name is specified to have the INTRINSIC attribute, and if the reference is consistent with the interface of that intrinsic procedure, the reference is to that intrinsic procedure.

NOTE 12.35

In the USE statement case, it is possible, because of the renaming facility, for the name in the reference to be different from the name of the intrinsic procedure.

(4) If (1), (2), and (3) do not apply, if the scoping unit has a host scoping unit, if the name is established to be generic in that host scoping unit, and if there is agreement between the scoping unit and the host scoping unit as to whether the name is a function name or a subroutine name, the name is resolved by applying the rules in this section to the host scoping unit.

12.4.4.2 Resolving procedure references to names established to be only specific

(1) If the scoping unit contains an interface body or EXTERNAL attribute specification for the name, if the scoping unit is a subprogram, and if the name is the name of a dummy argument of that subprogram, the dummy argument is a dummy procedure and the reference is to that dummy procedure. That is, the procedure invoked by executing that reference is the procedure supplied as the actual argument corresponding to that dummy procedure.

(2) If the scoping unit contains an interface body or EXTERNAL attribute specification for the name and if (1) does not apply, the reference is to an external procedure with that name.

(3) If the scoping unit contains a module subprogram, internal subprogram, or statement function that defines a procedure with the name, the reference is to the procedure so defined.

(4) If the scoping unit contains an INTRINSIC attribute specification for the name, the reference is to the intrinsic with that name.

(5) If the scoping unit contains a USE statement that makes a procedure accessible by the name, the reference is to that procedure.

NOTE 12.36

Because of the renaming facility of the USE statement, the name in the reference may be different from the original name of the procedure.

(6) If none of the above apply, the scoping unit shall have a host scoping unit, and the reference is resolved by applying the rules in this section to the host scoping unit.

12.4.4.3 Resolving procedure references to names not established

(1) If the scoping unit is a subprogram and if the name is the name of a dummy argument of that subprogram, the dummy argument is a dummy procedure and the reference is to that dummy procedure. That is, the procedure invoked by executing that reference is the procedure supplied as the actual argument corresponding to that dummy procedure.

(2) If (1) does not apply, if the name is the name of an intrinsic procedure, and if there is agreement between the reference and the status of the intrinsic procedure as being a function or subroutine, the reference is to that intrinsic procedure.

(3) If (1) and (2) do not apply, the reference is to an external procedure with that name.

12.4.5 Resolving type-bound procedure references

If the *binding-name* in a *procedure-designator* (R1219) is that of a specific type-bound procedure, the procedure referenced is the one bound to that name in the dynamic type of the *data-ref*.

If the *binding-name* in a *procedure-designator* is that of a generic type bound procedure, the generic binding with that name in the declared type of the *data-ref* is used to select a specific binding:

(1) If the reference is consistent with one of the specific bindings of that generic binding, that specific binding is selected.

(2) Otherwise, the reference shall be consistent with an elemental reference to one of the specific bindings of that generic binding; that specific binding is selected.

The reference is to the procedure bound to the same name as the selected specific binding in the dynamic type of the *data-ref*.

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 shall include the intrinsic procedures described in Section 13, but may include others. However, a standard-conforming program shall not make use of intrinsic procedures other than those described in Section 13.

12.5.2 Procedures defined by subprograms

When a procedure defined by a subprogram is invoked, an instance (12.5.2.3) of the 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 Function subprogram

A **function subprogram** is a subprogram that has a FUNCTION statement as its first statement.

R1223 *function-subprogram* **is** *function-stmt*
    [*specification-part*]
    [*execution-part*]
    [*internal-subprogram-part*]
    end-function-stmt

R1224 *function-stmt* **is** [*prefix*] FUNCTION *function-name* ■
    ■ ([*dummy-arg-name-list*]) [*su x*]

C1234 (R1224) If RESULT is specified, *result-name* shall not be the same as *function-name* and shall not be the same as the *entry-name* in any ENTRY statement in the subprogram.

C1235 (R1224) If RESULT is specified, the *function-name* shall not appear in any specification statement in the scoping unit of the function subprogram.

R1225 *proc-language-binding-spec* **is** *language-binding-spec*

C1236 (R1225) A *proc-language-binding-spec* with a NAME= specifier shall not be specified in the *function-stmt* or *subroutine-stmt* of an interface body for an abstract interface or a dummy procedure.

C1237 (R1225) A *proc-language-binding-spec* shall not be specified for an internal procedure.

C1238 (R1225) If *proc-language-binding-spec* is specified for a procedure, each of the procedure's dummy arguments shall be a nonoptional interoperable variable (15.2.4, 15.2.5) or an interoperable procedure (15.2.6). If *proc-language-binding-spec* is specified for a function, the function result shall be an interoperable variable.

R1226 *dummy-arg-name* **is** *name*

C1239 (R1226) A *dummy-arg-name* shall be the name of a dummy argument.

R1227 *prefix* **is** *prefix-spec* [*prefix-spec*] ...
R1228 *prefix-spec* **is** *declaration-type-spec*

or RECURSIVE
or PURE
or ELEMENTAL

C1240 (R1227) A *prefix* shall contain at most one of each *prefix-spec*.

C1241 (R1227) A *prefix* shall not specify both ELEMENTAL and RECURSIVE.

C1242 (R1227) A *prefix* shall not specify ELEMENTAL if *proc-language-binding-spec* appears in the *function-stmt* or *subroutine-stmt*.

R1229 *suffix* **is** *proc-language-binding-spec* [RESULT (*result-name*)]
or RESULT (*result-name*) [*proc-language-binding-spec*]

R1230 *end-function-stmt* **is** END [FUNCTION [*function-name*]]

C1243 (R1230) FUNCTION shall appear in the *end-function-stmt* of an internal or module function.

C1244 (R1223) An internal function subprogram shall not contain an ENTRY statement.

C1245 (R1223) An internal function subprogram shall not contain an *internal-subprogram-part*.

C1246 (R1230) If a *function-name* appears in the *end-function-stmt*, it shall be identical to the *function-name* specified in the *function-stmt*.

The name of the function is *function-name*.

The type and type parameters (if any) of the result of the function defined by a function subprogram may be specified by a type specification in the FUNCTION statement or by the name of the result variable appearing in a type declaration statement in the declaration part of the function subprogram. They shall not be specified both ways. If they are not specified either way, they are determined by the implicit typing rules in force within the function subprogram. If the function result is an array, allocatable, or a pointer, this shall be specified by specifications of the name of the result variable within the function body. The specifications of the function result attributes, the specification of dummy argument attributes, and the information in the procedure heading collectively define the characteristics of the function (12.2).

The *prefix-spec* RECURSIVE shall appear if the function directly or indirectly invokes itself or a function defined by an ENTRY statement in the same subprogram. Similarly, RECURSIVE shall appear if a function defined by an ENTRY statement in the subprogram directly or indirectly invokes itself, another function defined by an ENTRY statement in that subprogram, or the function defined by the FUNCTION statement.

If RESULT is specified, the name of the result variable of the function is *result-name* and all occurrences of the function name in *execution-part* statements in the scoping unit refer to the function itself. If RESULT is not specified, the result variable is *function-name* and all occurrences of the function name in *execution-part* statements in the scoping unit are references to the result variable. The characteristics (12.2.2) of the function result are those of the result variable. On completion of execution of the function, the value returned is that of its result variable. If the function result is a pointer, the shape of the value returned by the function is determined by the shape of the result variable when the execution of the function is completed. If the result variable is not a pointer, its value shall be defined by the function. If the function result is a pointer, the function shall either associate a target with the result variable pointer or cause the association status of this pointer to become defined as disassociated.

NOTE 12.37

The result variable is similar to any other variable local to a function subprogram. Its existence begins when execution of the function is initiated and ends when execution of the function is

NOTE 12.37 (cont.)

terminated. However, because the final value of this variable is used subsequently in the evaluation of the expression that invoked the function, an implementation may wish to defer releasing the storage occupied by that variable until after its value has been used in expression evaluation.

If the *prefix-spec* PURE or ELEMENTAL appears, the subprogram is a pure subprogram and shall meet the additional constraints of 12.6.

If the *prefix-spec* ELEMENTAL appears, the subprogram is an elemental subprogram and shall meet the additional constraints of 12.7.1.

NOTE 12.38

An example of a recursive function is:

```
RECURSIVE FUNCTION CUMM_SUM (ARRAY) RESULT (C_SUM)
   REAL, INTENT (IN), DIMENSION (:) :: ARRAY
   REAL, DIMENSION (SIZE (ARRAY)) ::C_SUM
   INTEGER N
   N = SIZE (ARRAY)
   IF (N <= 1) THEN
      C_SUM = ARRAY
   ELSE
      N = N / 2
      C_SUM (:N) = CUMM_SUM (ARRAY (:N))
      C_SUM (N+1:) = C_SUM (N) + CUMM_SUM (ARRAY (N+1:))
   END IF
END FUNCTION CUMM_SUM
```

NOTE 12.39

The following is an example of the declaration of an interface body with the BIND attribute, and a reference to the procedure declared.

```
USE, INTRINSIC :: ISO_C_BINDING

INTERFACE
  FUNCTION JOE (I, J, R), BIND(C,NAME="FrEd")
    USE, INTRINSIC :: ISO_C_BINDING
    INTEGER(C_INT) :: JOE
    INTEGER(C_INT), VALUE :: I, J
    REAL(C_FLOAT), VALUE :: R
  END FUNCTION JOE
END INTERFACE

INT = JOE(1_C_INT, 3_C_INT, 4.0_C_FLOAT)
END PROGRAM
```

The invocation of the function JOE results in a reference to a function with a binding label `"FrEd"`. FrEd may be a C function described by the C prototype

```
  int FrEd(int n, int m, float x);
```

12.5.2.2 Subroutine subprogram

A **subroutine subprogram** is a subprogram that has a SUBROUTINE statement as its first statement.

R1231 *subroutine-subprogram* **is** *subroutine-stmt*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-subroutine-stmt

R1232 *subroutine-stmt* **is** [*prefix*] SUBROUTINE *subroutine-name* ■
■ [([*dummy-arg-list*]) [*proc-language-binding-spec*]]

C1247 (R1232) The *prefix* of a *subroutine-stmt* shall not contain a *declaration-type-spec*.

R1233 *dummy-arg* **is** *dummy-arg-name*
or *

R1234 *end-subroutine-stmt* **is** END [SUBROUTINE [*subroutine-name*]]

C1248 (R1234) SUBROUTINE shall appear in the *end-subroutine-stmt* of an internal or module subroutine.

C1249 (R1231) An internal subroutine subprogram shall not contain an ENTRY statement.

C1250 (R1231) An internal subroutine subprogram shall not contain an *internal-subprogram-part*.

C1251 (R1234) If a *subroutine-name* appears in the *end-subroutine-stmt*, it shall be identical to the *subroutine-name* specified in the *subroutine-stmt*.

The name of the subroutine is *subroutine-name*.

The *prefix-spec* RECURSIVE shall appear if the subroutine directly or indirectly invokes itself or a subroutine defined by an ENTRY statement in the same subprogram. Similarly, RECURSIVE shall appear if a subroutine defined by an ENTRY statement in the subprogram directly or indirectly invokes itself, another subroutine defined by an ENTRY statement in that subprogram, or the subroutine defined by the SUBROUTINE statement.

If the *prefix-spec* PURE or ELEMENTAL appears, the subprogram is a pure subprogram and shall meet the additional constraints of 12.6.

If the *prefix-spec* ELEMENTAL appears, the subprogram is an elemental subprogram and shall meet the additional constraints of 12.7.1.

12.5.2.3 Instances of a subprogram

When a function or subroutine defined by a subprogram is invoked, an **instance** of that subprogram is created. When a statement function is invoked, an instance of that statement function is created.

Each instance has an independent sequence of execution and an independent set of dummy arguments and local unsaved data objects. If an internal procedure or statement function in the subprogram is invoked directly from an instance of the subprogram or from an internal subprogram or statement function that has access to the entities of that instance, the created instance of the internal subprogram or statement function also has access to the entities of that instance of the host subprogram.

All other entities are shared by all instances of the subprogram.

NOTE 12.40

The value of a saved data object appearing in one instance may have been defined in a previous instance or by initialization in a DATA statement or type declaration statement.

12.5.2.4 ENTRY statement

An **ENTRY statement** permits a procedure reference to begin with a particular executable statement within the function or subroutine subprogram in which the ENTRY statement appears.

R1235 *entry-stmt* **is** ENTRY *entry-name* [([*dummy-arg-list*]) [*su x*]]

C1252 (R1235) If RESULT is specified, the *entry-name* shall not appear in any specification or type-declaration statement in the scoping unit of the function program.

C1253 (R1235) An *entry-stmt* shall appear only in an *external-subprogram* or *module-subprogram*. An *entry-stmt* shall not appear within an *executable-construct*.

C1254 (R1235) RESULT shall appear only if the *entry-stmt* is in a function subprogram.

C1255 (R1235) Within the subprogram containing the *entry-stmt*, the *entry-name* shall not appear as a dummy argument in the FUNCTION or SUBROUTINE statement or in another ENTRY statement nor shall it appear in an EXTERNAL, INTRINSIC, or PROCEDURE statement.

C1256 (R1235) A *dummy-arg* shall not be an alternate return indicator if the ENTRY statement is in a function subprogram.

C1257 (R1235) If RESULT is specified, *result-name* shall not be the same as the *function-name* in the FUNCTION statement and shall not be the same as the *entry-name* in any ENTRY statement in the subprogram.

Optionally, a subprogram may have one or more ENTRY statements.

If the ENTRY statement is in a function subprogram, an additional function is defined by that subprogram. The name of the function is *entry-name* and the name of its result variable is *result-name* or is *entry-name* if no *result-name* is provided. The characteristics of the function result are specified by specifications of the result variable. The dummy arguments of the function are those specified in the ENTRY statement. If the characteristics of the result of the function named in the ENTRY statement are the same as the characteristics of the result of the function named in the FUNCTION statement, their result variables identify the same variable, although their names need not be the same. Otherwise, they are storage associated and shall all be nonpointer, nonallocatable scalars of type default integer, default real, double precision real, default complex, or default logical.

If the ENTRY statement is 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 in the ENTRY statement.

The order, number, types, kind type parameters, and names of the dummy arguments in an ENTRY statement may differ from the order, number, types, kind type parameters, and names of the dummy arguments in the FUNCTION or SUBROUTINE statement in the containing subprogram.

Because an ENTRY statement defines an additional function or an additional subroutine, it is referenced in the same manner as any other function or subroutine (12.4).

In a subprogram, a name that appears as a dummy argument in an ENTRY statement shall not appear in an executable statement preceding that ENTRY statement, unless it also appears in a FUNCTION, SUBROUTINE, or ENTRY statement that precedes the executable statement.

In a subprogram, a name that appears as a dummy argument in an ENTRY statement shall not appear in the expression of a statement function unless the name is also a dummy argument of the statement function, appears in a FUNCTION or SUBROUTINE statement, or appears in an ENTRY statement that precedes the statement function statement.

If a dummy argument appears in an executable statement, the execution of the executable statement is permitted during the execution of a reference to the function or subroutine only if the dummy argument appears in the dummy argument list of the procedure name referenced.

If a dummy argument is used in a specification expression to specify an array bound or character length of an object, the appearance of the object in a statement that is executed during a procedure reference is permitted only if the dummy argument appears in the dummy argument list of the procedure name referenced and it is present (12.4.1.6).

A scoping unit containing a reference to a procedure defined by an ENTRY statement may have access to an interface body for the procedure. The procedure header for the interface body shall be a FUNCTION statement for an entry in a function subprogram and shall be a SUBROUTINE statement for an entry in a subroutine subprogram.

The keyword RECURSIVE is not used in an ENTRY statement. Instead, the presence or absence of RECURSIVE in the initial SUBROUTINE or FUNCTION statement controls whether the procedure defined by an ENTRY statement is permitted to reference itself.

The keyword PURE is not used in an ENTRY statement. Instead, the procedure defined by an ENTRY statement is pure if and only if PURE or ELEMENTAL is specified in the SUBROUTINE or FUNCTION statement.

The keyword ELEMENTAL is not used in an ENTRY statement. Instead, the procedure defined by an ENTRY statement is elemental if and only if ELEMENTAL is specified in the SUBROUTINE or FUNCTION statement.

12.5.2.5 RETURN statement

R1236 *return-stmt* **is** RETURN [*scalar-int-expr*]

C1258 (R1236) The *return-stmt* shall be in the scoping unit of a function or subroutine subprogram.

C1259 (R1236) The *scalar-int-expr* is allowed only in the scoping unit of a subroutine subprogram.

Execution of the RETURN statement completes execution of the instance of the subprogram in which it appears. If the expression appears 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 nth 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-function-stmt* or *end-subroutine-stmt* is equivalent to executing a RETURN statement with no expression.

12.5.2.6 CONTAINS statement

R1237 *contains-stmt* **is** CONTAINS

The CONTAINS statement separates the body of a main program, module, or subprogram from any internal or module subprograms it may contain, or it introduces the type-bound procedure part of a derived-type definition (4.5.1). The CONTAINS statement is not executable.

12.5.3 Definition and invocation of procedures by means other than Fortran

A procedure may be defined by means other than Fortran. The interface of a procedure defined by means other than Fortran may be specified in an interface block. If the interface of such a procedure does not have a *proc-language-binding-spec*, the means by which the procedure is defined are processor dependent. A reference to such a procedure is made as though it were defined by an external subprogram.

If the interface of a procedure has a *proc-language-binding-spec*, the procedure is interoperable (15.4).

Interoperation with C functions is described in 15.4.

NOTE 12.41

For explanatory information on definition of procedures by means other than Fortran, see section C.9.2.

12.5.4 Statement function

A statement function is a function defined by a single statement.

R1238 *stmt-function-stmt* **is** *function-name* ([*dummy-arg-name-list*]) = *scalar-expr*

C1260 (R1238) The *primaries* of the *scalar-expr* shall be constants (literal and named), references to variables, references to functions and function dummy procedures, and intrinsic operations. If *scalar-expr* contains a reference to a function or a function dummy procedure, the reference shall not require an explicit interface, the function shall not require an explicit interface unless it is an intrinsic, the function shall not be a transformational intrinsic, and the result shall be scalar. If an argument to a function or a function dummy procedure is an array, it shall be an array name. If a reference to a statement function appears in *scalar-expr*, its definition shall have been provided earlier in the scoping unit and shall not be the name of the statement function being defined.

C1261 (R1238) Named constants in *scalar-expr* shall have been declared earlier in the scoping unit or made accessible by use or host association. If array elements appear in *scalar-expr*, the array shall have been declared as an array earlier in the scoping unit or made accessible by use or host association.

C1262 (R1238) If a *dummy-arg-name*, variable, function reference, or dummy function reference is typed by the implicit typing rules, its appearance in any subsequent type declaration statement shall confirm this implied type and the values of any implied type parameters.

C1263 (R1238) The *function-name* and each *dummy-arg-name* shall be specified, explicitly or implicitly, to be scalar.

C1264 (R1238) A given *dummy-arg-name* shall not appear more than once in any *dummy-arg-name-list*.

C1265 (R1238) Each variable reference in *scalar-expr* may be either a reference to a dummy argument of the statement function or a reference to a variable accessible in the same scoping unit as the statement function statement.

The definition of a statement function with the same name as an accessible entity from the host shall be preceded by the declaration of its type in a type declaration statement.

The dummy arguments have a scope of the statement function statement. Each dummy argument has the same type and type parameters as the entity of the same name in the scoping unit containing the statement function.

A statement function shall not be supplied as a procedure argument.

The value of a statement function reference is obtained by evaluating the expression using the values of the actual arguments for the values of the corresponding dummy arguments and, if necessary, converting the result to the declared type and type attributes of the function.

A function reference in the scalar expression shall not cause a dummy argument of the statement function to become redefined or undefined.

12.6 Pure procedures

A **pure procedure** is

(1) A pure intrinsic function (13.1),
(2) A pure intrinsic subroutine (13.1),
(3) Defined by a pure subprogram, or
(4) A statement function that references only pure functions.

A pure subprogram is a subprogram that has the *prefix-spec* PURE or ELEMENTAL. The following additional constraints apply to pure subprograms.

C1266 The *specification-part* of a pure function subprogram shall specify that all its nonpointer dummy data objects have INTENT(IN).

C1267 The *specification-part* of a pure subroutine subprogram shall specify the intents of all its nonpointer dummy data objects.

C1268 A local variable declared in the *specification-part* or *internal-subprogram-part* of a pure subprogram shall not have the SAVE attribute.

NOTE 12.42

Variable initialization in a *type-declaration-stmt* or a *data-stmt* implies the SAVE attribute; therefore, such initialization is also disallowed.

C1269 The *specification-part* of a pure subprogram shall specify that all its dummy procedures are pure.

C1270 If a procedure that is neither an intrinsic procedure nor a statement function is used in a context that requires it to be pure, then its interface shall be explicit in the scope of that use. The interface shall specify that the procedure is pure.

C1271 All internal subprograms in a pure subprogram shall be pure.

C1272 In a pure subprogram any designator with a base object that is in common or accessed by host or use association, is a dummy argument of a pure function, is a dummy argument with INTENT (IN) of a pure subroutine, or an object that is storage associated with any such variable, shall not be used in the following contexts:

(1) In a variable definition context(16.5.7);
(2) As the *data-target* in a *pointer-assignment-stmt*;
(3) As the *expr* corresponding to a component with the POINTER attribute in a *structure-constructor*.
(4) As the *expr* of an intrinsic assignment statement in which the *variable* is of a derived type if the derived type has a pointer component at any level of component selection; or

NOTE 12.43

This requires that processors be able to determine if entities with the PRIVATE attribute or with private components have a pointer component.

(5) As an actual argument associated with a dummy argument with INTENT (OUT) or INTENT (INOUT) or with the POINTER attribute.

C1273 Any procedure referenced in a pure subprogram, including one referenced via a defined operation, assignment, or finalization, shall be pure.

C1274 A pure subprogram shall not contain a *print-stmt*, *open-stmt*, *close-stmt*, *backspace-stmt*, *endfile-stmt*, *rewind-stmt*, *flush-stmt*, *wait-stmt*, or *inquire-stmt*.

C1275 A pure subprogram shall not contain a *read-stmt* or *write-stmt* whose *io-unit* is a *file-unit-number* or *.

C1276 A pure subprogram shall not contain a *stop-stmt*.

NOTE 12.44

The above constraints are designed to guarantee that a pure procedure is free from side effects (modifications of data visible outside the procedure), which means that it is safe to reference it in constructs such as a FORALL *assignment-stmt* where there is no explicit order of evaluation.

The constraints on pure subprograms may appear complicated, but it is not necessary for a programmer to be intimately familiar with them. From the programmer's point of view, these constraints can be summarized as follows: a pure subprogram shall not contain any operation that could conceivably result in an assignment or pointer assignment to a common variable, a variable accessed by use or host association, or an INTENT (IN) dummy argument; nor shall a pure subprogram contain any operation that could conceivably perform any external file input/output or STOP operation. Note the use of the word conceivably; it is not sufficient for a pure subprogram merely to be side-effect free in practice. For example, a function that contains an assignment to a global variable but in a block that is not executed in any invocation of the function is nevertheless not a pure function. The exclusion of functions of this nature is required if strict compile-time checking is to be used.

It is expected that most library procedures will conform to the constraints required of pure procedures, and so can be declared pure and referenced in FORALL statements and constructs and within user-defined pure procedures.

NOTE 12.45

Pure subroutines are included to allow subroutine calls from pure procedures in a safe way, and to allow *forall-assignment-stmt*s to be defined assignments. The constraints for pure subroutines are based on the same principles as for pure functions, except that side effects to INTENT (OUT), INTENT (INOUT), and pointer dummy arguments are permitted.

12.7 Elemental procedures

12.7.1 Elemental procedure declaration and interface

An **elemental procedure** is an elemental intrinsic procedure or a procedure that is defined by an elemental subprogram.

An elemental subprogram has the *prefix-spec* ELEMENTAL. An elemental subprogram is a pure subprogram. The PURE *prefix-spec* need not appear; it is implied by the ELEMENTAL *prefix-spec*. The following additional constraints apply to elemental subprograms.

C1277 All dummy arguments of an elemental procedure shall be scalar dummy data objects and shall not have the POINTER or ALLOCATABLE attribute.

C1278 The result variable of an elemental function shall be scalar and shall not have the POINTER or

ALLOCATABLE attribute.

C1279 In the scoping unit of an elemental subprogram, an object designator with a dummy argument as the base object shall not appear in a *specification-expr* except as the argument to one of the intrinsic functions BIT_SIZE, KIND, LEN, or the numeric inquiry functions (13.5.6).

NOTE 12.46

An elemental subprogram is a pure subprogram and all of the constraints for pure subprograms also apply.

NOTE 12.47

The restriction on dummy arguments in specification expressions is imposed primarily to facilitate optimization. An example of usage that is not permitted is

```
ELEMENTAL REAL FUNCTION F (A, N)
   REAL, INTENT (IN) :: A
   INTEGER, INTENT (IN) :: N
   REAL :: WORK_ARRAY(N)   ! Invalid
   ...
END FUNCTION F
```

An example of usage that is permitted is

```
ELEMENTAL REAL FUNCTION F (A)
   REAL, INTENT (IN) :: A
   REAL (SELECTED_REAL_KIND (PRECISION (A)*2)) :: WORK
   ...
END FUNCTION F
```

12.7.2 Elemental function actual arguments and results

If a generic name or a specific name is used to reference an elemental function, the shape of the result is the same as the shape of the actual argument with the greatest rank. If there are no actual arguments or the actual arguments are all scalar, the result is scalar. For those elemental functions that have more than one argument, all actual arguments shall be conformable. In the array case, the values of the elements, if any, of the result are the same as would have been obtained if the scalar function had been applied separately, in any order, to corresponding elements of each array actual argument.

NOTE 12.48

An example of an elemental reference to the intrinsic function MAX:

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

```
MAX (X(I, J),  0.0, Y(I, J)), I = 1, 2, ..., M, J = 1,2, ..., N
```

12.7.3 Elemental subroutine actual arguments

An elemental subroutine is one that has only scalar dummy arguments, but may have array actual arguments. In a reference to an elemental subroutine, either all actual arguments shall be scalar, or all actual arguments associated with INTENT (OUT) and INTENT (INOUT) dummy arguments shall be arrays of the same shape and the remaining actual arguments shall be conformable with them. In the case that the actual arguments associated with INTENT (OUT) and INTENT (INOUT) dummy arguments are arrays, the values of the elements, if any, of the results are the same as would be obtained if the subroutine had been applied separately, in any order, to corresponding elements of each array actual argument.

In a reference to the intrinsic subroutine MVBITS, the actual arguments corresponding to the TO and FROM dummy arguments may be the same variable and may be associated scalar variables or associated array variables all of whose corresponding elements are associated. Apart from this, the actual arguments in a reference to an elemental subroutine must satisfy the restrictions of 12.4.1.7.

Section 13: Intrinsic procedures and modules

There are four classes of intrinsic procedures: inquiry functions, elemental functions, transformational functions, and subroutines. Some intrinsic subroutines are elemental.

There are three sets of standard intrinsic modules: a Fortran environment module (13.8.2), modules to support exception handling and IEEE arithmetic, and a module to support interoperability with the C programming language. The later two sets of modules are described in sections 14 and 15, respectively.

13.1 Classes of intrinsic procedures

An **inquiry function** is one whose result depends on the properties of one or more of its arguments instead of their values; in fact, these argument values may be undefined. Unless the description of an inquiry function states otherwise, these arguments are permitted to be unallocated allocatables or pointers that are not associated. An **elemental intrinsic function** is one that is specified for scalar arguments, but may be applied to array arguments as described in 12.7. All other intrinsic functions are **transformational functions**; they almost all have one or more array arguments or an array result. All standard intrinsic functions are pure.

The subroutine MOVE_ALLOC and the elemental subroutine MVBITS are pure. No other standard intrinsic subroutine is pure.

NOTE 13.1

NOTE 13.2

The text CMPLX (X [, Y, KIND]) indicates that Y and KIND are both optional arguments. Valid reference forms include CMPLX(*x*), CMPLX(*x*, *y*), CMPLX(*x*, KIND=*kind*), CMPLX(*x*, *y*, *kind*), and CMPLX(KIND=*kind*, X=*x*, Y=*y*).

NOTE 13.3

Some intrinsic procedures impose additional requirements on their optional arguments. For example, SELECTED_REAL_KIND requires that at least one of its optional arguments be present, and RANDOM_SEED requires that at most one of its optional arguments be present.

The dummy arguments of the specific intrinsic procedures in 13.6 have INTENT(IN). The dummy arguments of the generic intrinsic procedures in 13.7 have INTENT(IN) if the intent is not stated explicitly.

The actual argument associated with an intrinsic function dummy argument named KIND shall be a scalar integer initialization expression and its value shall specify a representation method for the function result that exists on the processor.

Intrinsic subroutines that assign values to arguments of type character do so in accordance with the rules of intrinsic assignment (7.4.1.3).

13.2.1 The shape of array arguments

Unless otherwise specified, the inquiry intrinsic functions accept array arguments for which the shape need not be defined. The shape of array arguments to transformational and elemental intrinsic functions shall be defined.

13.2.2 Mask arguments

Some array intrinsic functions have an optional MASK argument of type logical 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 the time 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.

13.3 Bit model

The bit manipulation procedures are ten elemental functions and one elemental subroutine. Logical operations on bits are provided by the elemental functions IOR, IAND, NOT, and IEOR; shift operations are provided by the elemental functions ISHFT and ISHFTC; bit subfields may be referenced by the elemental function IBITS and by the elemental subroutine MVBITS; single-bit processing is provided by the elemental functions BTEST, IBSET, and IBCLR.

For the purposes of these procedures, a bit is defined to be a binary digit w located at position k of a nonnegative integer scalar object based on a model nonnegative integer defined by

$$j = \sum_{k=0}^{z-1} w_k \times 2^k$$

and for which w_k may have the value 0 or 1. An example of a model number compatible with the examples used in 13.4 would have $z = 32$, thereby defining a 32-bit integer.

1 An inquiry function BIT_SIZE is available to determine the parameter z of the model.

2 Effectively, this model defines an integer object to consist of z bits in sequence numbered from right to left from 0 to $z -$

13.5 Standard generic intrinsic procedures

For all of the standard intrinsic procedures, the arguments shown are the names that shall be used for argument keywords if the keyword form is used for actual arguments.

NOTE 13.5

For example, a reference to CMPLX may be written in the form CMPLX (A, B, M) or in the form CMPLX (Y = B, KIND = M, X = A).

NOTE 13.6

Many of the argument keywords have names that are indicative of their usage. For example:

KIND	Describes the kind type parameter of the result
STRING, STRING_A	An arbitrary character string
BACK	Controls the direction of string scan (forward or backward)
MASK	A mask that may be applied to the arguments
DIM	A selected dimension of an array argument

13.5.1 Numeric functions

ABS (A)	Absolute value
AIMAG (Z)	Imaginary part of a complex number
AINT (A [, KIND])	Truncation to whole number
ANINT (A [, KIND])	Nearest whole number
CEILING (A [, KIND])	Least integer greater than or equal to number
CMPLX (X [, Y, KIND])	Conversion to complex type
CONJG (Z)	Conjugate of a complex number
DBLE (A)	Conversion to double precision real type
DIM (X, Y)	Positive difference
DPROD (X, Y)	Double precision real product
FLOOR (A [, KIND])	Greatest integer less than or equal to number
INT (A [, KIND])	Conversion to integer type
MAX (A1, A2 [, A3,...])	Maximum value
MIN (A1, A2 [, A3,...])	Minimum value
MOD (A, P)	Remainder function
MODULO (A, P)	Modulo function
NINT (A [, KIND])	Nearest integer
REAL (A [, KIND])	Conversion to real type
SIGN (A, B)	Transfer of sign

13.5.2 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

13.5.3 Character functions

ACHAR (I [, KIND])	Character in given position in ASCII collating sequence
ADJUSTL (STRING)	Adjust left
ADJUSTR (STRING)	Adjust right
CHAR (I [, KIND])	Character in given position in processor collating sequence
IACHAR (C [, KIND])	Position of a character in ASCII collating sequence
ICHAR (C [, KIND])	Position of a character in processor collating sequence
INDEX (STRING, SUBSTRING [, BACK, KIND])	Starting position of a substring
LEN_TRIM (STRING [, KIND])	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
MAX (A1, A2 [, A3,...])	Maximum value
MIN (A1, A2 [, A3,...])	Minimum value
REPEAT (STRING, NCOPIES)	Repeated concatenation
SCAN (STRING, SET [, BACK, KIND])	Scan a string for a character in a set
TRIM (STRING)	Remove trailing blank characters
VERIFY (STRING, SET [, BACK, KIND])	Verify the set of characters in a string

13.5.4 Kind functions

KIND (X)	Kind type parameter value
SELECTED_CHAR_KIND (NAME)	Character kind type parameter value, given character set name
SELECTED_INT_KIND (R)	Integer kind type parameter value, given range
SELECTED_REAL_KIND ([P, R])	Real kind type parameter value, given precision and range

13.5.5 Miscellaneous type conversion functions

LOGICAL (L [, KIND])	Convert between objects of type logical with different kind type parameters
TRANSFER (SOURCE, MOLD [, SIZE])	Treat first argument as if of type of second argument

13.5.6 Numeric inquiry functions

DIGITS (X)	Number of significant digits of the model
EPSILON (X)	Number that is almost negligible compared to one
HUGE (X)	Largest number of the model
MAXEXPONENT (X)	Maximum exponent of the model

MINEXPONENT (X)	Minimum exponent of the model
PRECISION (X)	Decimal precision
RADIX (X)	Base of the model
RANGE (X)	Decimal exponent range
TINY (X)	Smallest positive number of the model

13.5.7 Array inquiry functions

LBOUND (ARRAY [, DIM, KIND])	Lower dimension bounds of an array
SHAPE (SOURCE [, KIND])	Shape of an array or scalar
SIZE (ARRAY [, DIM, KIND])	Total number of elements in an array
UBOUND (ARRAY [, DIM, KIND])	Upper dimension bounds of an array

13.5.8 Other inquiry functions

ALLOCATED (ARRAY) or ALLOCATED (SCALAR)	Allocation status
ASSOCIATED (POINTER [, TARGET])	Association status inquiry or comparison
BIT_SIZE (I)	Number of bits of the model
EXTENDS_TYPE_OF (A, MOLD)	Same dynamic type or an extension
LEN (STRING [, KIND])	Length of a character entity
NEW_LINE (A)	Newline character
PRESENT (A)	Argument presence
SAME_TYPE_AS (A, B)	Same dynamic type

13.5.9 Bit manipulation procedures

BTEST (I, POS)	Bit testing
IAND (I, J)	Bitwise AND
IBCLR (I, POS)	Clear bit
IBITS (I, POS, LEN)	Bit extraction
IBSET (I, POS)	Set bit
IEOR (I, J)	Exclusive OR
IOR (I, J)	Inclusive OR
ISHFT (I, SHIFT)	Logical shift
ISHFTC (I, SHIFT [, SIZE])	Circular shift
MVBITS (FROM, FROMPOS, LEN, TO, TOPOS)	Copies bits from one integer to another
NOT (I)	Bitwise complement

13.5.10 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
SET_EXPONENT (X, I)	Set exponent part of a number
SPACING (X)	Absolute spacing of model numbers near given number

13.5.11 Vector and matrix multiply functions

DOT_PRODUCT (VECTOR_A, VECTOR_B)	Dot product of two rank-one arrays
MATMUL (MATRIX_A, MATRIX_B)	Matrix multiplication

13.5.12 Array reduction functions

ALL (MASK [, DIM])	True if all values are true
ANY (MASK [, DIM])	True if any value is true
COUNT (MASK [, DIM, KIND])	Number of true elements in an array
MAXVAL (ARRAY, DIM [, MASK]) or MAXVAL (ARRAY [, MASK])	Maximum value in an array
MINVAL (ARRAY, DIM [, MASK]) or MINVAL (ARRAY [, MASK])	Minimum value in an array
PRODUCT (ARRAY, DIM [, MASK]) or PRODUCT (ARRAY [, MASK])	Product of array elements
SUM (ARRAY, DIM [, MASK]) or SUM (ARRAY [, MASK])	Sum of array elements

13.5.13 Array construction functions

CSHIFT (ARRAY, SHIFT [, DIM])	Circular shift
EOSHIFT (ARRAY, SHIFT [, BOUNDARY, DIM])	End-off shift
MERGE (TSOURCE, FSOURCE, MASK)	Merge under mask
PACK (ARRAY, MASK [, VECTOR])	Pack an array into an array of rank one under a mask
RESHAPE (SOURCE, SHAPE[, PAD, ORDER])	Reshape an array
SPREAD (SOURCE, DIM, NCOPIES)	Replicates array by adding a dimension
TRANSPOSE (MATRIX)	Transpose of an array of rank two
UNPACK (VECTOR, MASK, FIELD)	Unpack an array of rank one into an array under a mask

13.5.14 Array location functions

MAXLOC (ARRAY, DIM [, MASK, KIND]) or MAXLOC (ARRAY [, MASK, KIND])	Location of a maximum value in an array
MINLOC (ARRAY, DIM [, MASK, KIND]) or MINLOC (ARRAY [, MASK, KIND])	Location of a minimum value in an array

13.5.15 Null function

NULL ([MOLD])	Returns disassociated or unallocated result

13.5.16 Allocation transfer procedure

MOVE_ALLOC (FROM, TO)	Moves an allocation from one allocatable object to another

13.5.17 Random number subroutines

RANDOM_NUMBER (HARVEST)	Returns pseudorandom number

RANDOM_SEED ([SIZE, PUT, GET])	Initializes or restarts the pseudorandom number generator

13.5.18 System environment procedures

COMMAND_ARGUMENT_COUNT ()	Number of command arguments
CPU_TIME (TIME)	Obtain processor time
DATE_AND_TIME ([DATE, TIME, ZONE, VALUES])	Obtain date and time
GET_COMMAND ([COMMAND, LENGTH, STATUS])	Returns entire command
GET_COMMAND_ARGUMENT (NUMBER [, VALUE, LENGTH, STATUS])	Returns a command argument
GET_ENVIRONMENT_VARIABLE (NAME [, VALUE, LENGTH, STATUS, TRIM_NAME])	Obtain the value of an environment variable
IS_IOSTAT_END (I)	Test for end-of-file value
IS_IOSTAT_EOR (I)	Test for end-of-record value
SYSTEM_CLOCK ([COUNT, COUNT_RATE, COUNT_MAX])	Obtain data from the system clock

13.6 Specific names for standard intrinsic functions

Except for AMAX0, AMIN0, MAX1, and MIN1, the result type of the specific function is the same as the result type of the corresponding generic function would be if it were invoked with the same arguments as the specific function.

	Specific Name	Generic Name	Argument Type
	ABS	ABS	default real
	ACOS	ACOS	default real
	AIMAG	AIMAG	default complex
	AINT	AINT	default real
	ALOG	LOG	default real
	ALOG10	LOG10	default real
•	AMAX0 (. . .)	REAL (MAX (. . .))	default integer
•	AMAX1	MAX	default real
•	AMIN0 (. . .)	REAL (MIN (. . .))	default integer
•	AMIN1	MIN	default real
	AMOD	MOD	default real
	ANINT	ANINT	default real
	ASIN	ASIN	default real
	ATAN	ATAN	default real
	ATAN2	ATAN2	default real
	CABS	ABS	default complex
	CCOS	COS	default complex
	CEXP	EXP	default complex
	CHAR	CHAR	default integer
	CLOG	LOG	default complex
	CONJG	CONJG	default complex
	COS	COS	default real
	COSH	COSH	default real
	CSIN	SIN	default complex
	CSQRT	SQRT	default complex

	Specific Name	Generic Name	Argument Type
	DABS	ABS	double precision real
	DACOS	ACOS	double precision real
	DASIN	ASIN	double precision real
	DATAN	ATAN	double precision real
	DATAN2	ATAN2	double precision real
	DCOS	COS	double precision real
	DCOSH	COSH	double precision real
	DDIM	DIM	double precision real
	DEXP	EXP	double precision real
	DIM	DIM	default real
	DINT	AINT	double precision real
	DLOG	LOG	double precision real
	DLOG10	LOG10	double precision real
•	DMAX1	MAX	double precision real
•	DMIN1	MIN	double precision real
	DMOD	MOD	double precision real
	DNINT	ANINT	double precision real
	DPROD	DPROD	default real
	DSIGN	SIGN	double precision real
	DSIN	SIN	double precision real
	DSINH	SINH	double precision real
	DSQRT	SQRT	double precision real
	DTAN	TAN	double precision real
	DTANH	TANH	double precision real
	EXP	EXP	default real
•	FLOAT	REAL	default integer
	IABS	ABS	default integer
•	ICHAR	ICHAR	default character
	IDIM	DIM	default integer
•	IDINT	INT	double precision real
	IDNINT	NINT	double precision real
•	IFIX	INT	default real
	INDEX	INDEX	default character
•	INT	INT	default real
	ISIGN	SIGN	default integer
	LEN	LEN	default character
•	LGE	LGE	default character
•	LGT	LGT	default character
•	LLE	LLE	default character
•	LLT	LLT	default character
•	MAX0	MAX	default integer
•	MAX1 (. . .)	INT (MAX (. . .))	default real
•	MIN0	MIN	default integer
•	MIN1 (. . .)	INT (MIN (. . .))	default real
	MOD	MOD	default integer
	NINT	NINT	default real
•	REAL	REAL	default integer
	SIGN	SIGN	default real
	SIN	SIN	default real
	SINH	SINH	default real
•	SNGL	REAL	double precision real
	SQRT	SQRT	default real

Specific Name	Generic Name	Argument Type
TAN	TAN	default real
TANH	TANH	default real

A specific intrinsic function marked with a bullet (•) shall not be used as an actual argument or as a target in a procedure pointer assignment statement.

13.7 Specifications of the standard intrinsic procedures

Detailed specifications of the standard generic intrinsic procedures are provided here in alphabetical order.

The types and type parameters of standard intrinsic procedure arguments and function results are determined by these specifications. The "Argument(s)" paragraphs specify requirements on the actual arguments of the procedures. The result characteristics are sometimes specified in terms of the characteristics of dummy arguments. A program is prohibited from invoking an intrinsic procedure under circumstances where a value to be returned in a subroutine argument or function result is outside the range of values representable by objects of the specified type and type parameters, unless the intrinsic module IEEE_ARITHMETIC (section 14) is accessible and there is support for an infinite or a NaN result, as appropriate. If an infinite result is returned, the flag IEEE_OVERFLOW or IEEE_DIVIDE_BY_ZERO shall signal; if a NaN result is returned, the flag IEEE_INVALID shall signal. If all results are normal, these flags shall have the same status as when the intrinsic procedure was invoked.

13.7.1 ABS (A)

Description. Absolute value.

Class. Elemental function.

Argument. A shall be of type integer, real, or complex.

Result Characteristics. The same as A except that if A is complex, the result is real.

Result Value. If A is of type integer or real, 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).

13.7.2 ACHAR (I [, KIND])

Description. Returns the character in a specified position of the ASCII collating sequence. It is the inverse of the IACHAR function.

Class. Elemental function.

Arguments.

I	shall be of type integer.
KIND (optional)	shall be a scalar integer initialization expression.

Result Characteristics. Character of length one. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default character type.

Result Value. If I has a value in the range $0 \leq I \leq 127$, the result is the character in position I of the ASCII collating sequence, provided the processor is capable of representing that character in the character type of the result; otherwise, the result is processor dependent. ACHAR (IACHAR (C)) shall have the value C for any character C capable of representation in the default character type.

Example. ACHAR (88) has the value 'X'.

13.7.3 ACOS (X)

Description. Arccosine (inverse cosine) function.

Class. Elemental function.

Argument. X shall be of type real with a value that satisfies the inequality $|X| \leq 1$.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to arccos(X), expressed in radians. It lies in the range $0 \leq \text{ACOS}(X) \leq \pi$.

Example. ACOS (0.54030231) has the value 1.0 (approximately).

13.7.4 ADJUSTL (STRING)

Description. Adjust to the left, removing leading blanks and inserting trailing blanks.

Class. Elemental function.

Argument. STRING shall be of type character.

Result Characteristics. Character of the same length and kind type parameter 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 the value 'WORD '.

13.7.5 ADJUSTR (STRING)

Description. Adjust to the right, removing trailing blanks and inserting leading blanks.

Class. Elemental function.

Argument. STRING shall be of type character.

Result Characteristics. Character of the same length and kind type parameter 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 the value ' WORD'.

13.7.6 AIMAG (Z)

Description. Imaginary part of a complex number.

Class. Elemental function.

Argument. Z shall be of type complex.

Result Characteristics. Real with the same kind type parameter as Z.

Result Value. If Z has the value (x, y), the result has the value y.

Example. AIMAG ((2.0, 3.0)) has the value 3.0.

13.7.7 AINT (A [, KIND])

Description. Truncation to a whole number.

Class. Elemental function.

Arguments.

A	shall be of type real.
KIND (optional)	shall be a scalar integer initialization expression.

Result Characteristics. The result is of type real. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of A.

Result Value. If $|A| < 1$, AINT (A) has the value 0; if $|A| \geq 1$, AINT (A) has a value equal to 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.

Examples. AINT (2.783) has the value 2.0. AINT (–2.783) has the value –2.0.

13.7.8 ALL (MASK [, DIM])

Description. Determine whether all values are true in MASK along dimension DIM.

Class. Transformational function.

Arguments.

MASK	shall be of type logical. It shall not be scalar.
DIM (optional)	shall be scalar and of type integer with value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of MASK. The corresponding actual argument shall not be an optional dummy argument.

Result Characteristics. The result is of type logical with the same kind type parameter as MASK. It is scalar if DIM is absent; otherwise, the result has rank $n - 1$ and 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 ALL (MASK) has the value true if all elements of MASK are true or if MASK has size zero, and the result has value false if any element of MASK is false.

Case (ii): If MASK has rank one, ALL(MASK,DIM) is equal to ALL(MASK). Otherwise, the value of element (s_1, s_2, $\ldots$, $s_{\text{DIM}-1}$, $s_{\text{DIM}+1}$, $\ldots$, s_n) of ALL (MASK, DIM) is equal to ALL (MASK (s_1, s_2, $\ldots$, $s_{\text{DIM}-1}$, :, $s_{\text{DIM}+1}$, ..., 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 /= C, DIM = 1) is [true, false, false] and ALL (B /= C, DIM = 2) is [false, false].

13.7.9 ALLOCATED (ARRAY) or ALLOCATED (SCALAR)

Description. Indicate whether an allocatable variable is allocated.

Class. Inquiry function.

Arguments.

ARRAY	shall be an allocatable array.
SCALAR	shall be an allocatable scalar.

Result Characteristics. Default logical scalar.

Result Value. The result has the value true if the argument (ARRAY or SCALAR) is allocated and has the value false if the argument is unallocated.

13.7.10 ANINT (A [, KIND])

Description. Nearest whole number.

Class. Elemental function.

Arguments.

A	shall be of type real.
KIND (optional)	shall be a scalar integer initialization expression.

Result Characteristics. The result is of type real. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of A.

Result Value. The result is the integer nearest A, or if there are two integers equally near A, the result is whichever such integer has the greater magnitude.

Examples. ANINT (2.783) has the value 3.0. ANINT (–2.783) has the value –3.0.

13.7.11 ANY (MASK [, DIM])

Description. Determine whether any value is true in MASK along dimension DIM.

Class. Transformational function.

Arguments.

MASK	shall be of type logical. It shall not be scalar.
DIM (optional)	shall be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of MASK. The corresponding actual argument shall not be an optional dummy argument.

Result Characteristics. The result is of type logical with the same kind type parameter as MASK. It is scalar if DIM is absent; otherwise, the result has rank $n - 1$ and 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 ANY (MASK) has the value true if any element of MASK is true and has the value false if no elements are true or if MASK has size zero.

Case (ii): If MASK has rank one, ANY(MASK,DIM) is equal to ANY(MASK). Otherwise, the value of element $(s_1, s_2, \ldots, s_{\mathrm{DIM}-1}, s_{\mathrm{DIM}+1}, \ldots, s_n)$ of ANY(MASK, DIM) is equal to ANY(MASK $(s_1, s_2, \ldots, s_{\mathrm{DIM}-1}, :, s_{\mathrm{DIM}+1}, \ldots, s_n)$).

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 $\begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$ then ANY(B /= C, DIM = 1) is [true, false, true] and ANY(B /= C, DIM = 2) is [true, true].

13.7.12 ASIN (X)

Description. Arcsine (inverse sine) function.

Class. Elemental function.

Argument. X shall be of type real. Its value shall satisfy the inequality $|X| \leq 1$.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to arcsin(X), expressed in radians. It lies in the range $-\pi/2 \leq \mathrm{ASIN}(X) \leq \pi/2$.

Example. ASIN (0.84147098) has the value 1.0 (approximately).

13.7.13 ASSOCIATED (POINTER [, TARGET])

Description. Returns the association status of its pointer argument or indicates whether the pointer is associated with the target.

Class. Inquiry function.

Arguments.

POINTER	shall be a pointer. It may be of any type or may be a procedure pointer. Its pointer association status shall not be undefined.
TARGET (optional)	shall be allowable as the *data-target* or *proc-target* in a pointer assignment statement (7.4.2) in which POINTER is *pointer-object*. If TARGET is a pointer then its pointer association status shall not be undefined.

Result Characteristics. Default logical scalar.

Result Value.

Case (i): If TARGET is absent, the result is true if POINTER is associated with a target and false if it is not.

Case (ii): If TARGET is present and is a procedure, the result is true if POINTER is associated with TARGET.

Case (iii): If TARGET is present and is a procedure pointer, the result is true if POINTER and TARGET are associated with the same procedure. If either POINTER or TARGET is disassociated, the result is false.

Case (iv): If TARGET is present and is a scalar target, the result is true if TARGET is not a zero-sized storage sequence and the target associated with POINTER occupies the same storage units as TARGET. Otherwise, the result is false. If the POINTER is disassociated, the result is false.

Case (v): If TARGET is present and is an array target, the result is true if the target associated with POINTER and TARGET have the same shape, are neither of size zero nor arrays whose elements are zero-sized storage sequences, and occupy the same storage units in array element order. Otherwise, the result is false. If POINTER is disassociated, the result is false.

Case (vi): If TARGET is present and is a scalar pointer, the result is true if the target associated with POINTER and the target associated with TARGET are not zero-sized storage sequences and they occupy the same storage units. Otherwise, the result is false. If either POINTER or TARGET is disassociated, the result is false.

Case (vii): If TARGET is present and is an array pointer, the result is true if the target associated with POINTER and the target associated with TARGET have the same shape, are neither of size zero nor arrays whose elements are zero-sized storage sequences, and occupy the same storage units in array element order. Otherwise, the result is false. If either POINTER or TARGET is disassociated, the result is false.

Examples. ASSOCIATED (CURRENT, HEAD) is true if CURRENT is associated with the target HEAD. After the execution of

```
A_PART => A (:N)
```

ASSOCIATED (A_PART, A) is true if N is equal to UBOUND (A, DIM = 1). After the execution of

```
NULLIFY (CUR); NULLIFY (TOP)
```

ASSOCIATED (CUR, TOP) is false.

13.7.14 ATAN (X)

Description. Arctangent (inverse tangent) function.

Class. Elemental function.

Argument. X shall be of type real.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to arctan(X), expressed in radians, that lies in the range $-\pi/2 \leq \text{ATAN(X)} \leq \pi/2$.

Example. ATAN (1.5574077) has the value 1.0 (approximately).

13.7.15 ATAN2 (Y, X)

Description. Arctangent (inverse tangent) function. The result is the principal value of the argument of the nonzero complex number (X, Y).

Class. Elemental function.

Arguments.

Y shall be of type real.

X shall be of the same type and kind type parameter as Y. If Y has the value zero, X shall not have the value zero.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to the principal value of the argument of the complex number (X, Y), expressed in radians. It lies in the range $-\pi \leq$ 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 $= 0$ and X > 0, the result is Y. If Y $= 0$ and X < 0, then the result is π if Y is positive real zero or the processor cannot distinguish between positive and negative real zero, and $-\pi$ if Y is negative real zero. If Y < 0, the result is negative. If X $= 0$, the absolute value of the result is $\pi/2$.

Examples. ATAN2 (1.5574077, 1.0) has the value 1.0 (approximately). If Y has the value $\begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix}$ and X has the value $\begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}$, the value of ATAN2 (Y, X) is approximately $\begin{bmatrix} \frac{3\pi}{4} & \frac{\pi}{4} \\ \frac{-3\pi}{4} & -\frac{\pi}{4} \end{bmatrix}$.

13.7.16 BIT_SIZE (I)

Description. Returns the number of bits z defined by the model of 13.3.

Class. Inquiry function.

Argument. I shall be of type integer. It may be a scalar or an array.

Result Characteristics. Scalar integer with the same kind type parameter as I.

Result Value. The result has the value of the number of bits z of the model integer defined for bit manipulation contexts in 13.3.

Example. BIT_SIZE (1) has the value 32 if z of the model is 32.

13.7.17 BTEST (I, POS)

Description. Tests a bit of an integer value.

Class. Elemental function.

Arguments.

I shall be of type integer.

POS shall be of type integer. It shall be nonnegative and be less than BIT_SIZE (I).

Result Characteristics. Default logical.

Result Value. The result has the value true if bit POS of I has the value 1 and has the value false if bit POS of I has the value 0. The model for the interpretation of an integer value as a sequence of bits is in 13.3.

Examples. BTEST (8, 3) has the value true. If A has the value $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, the value of

BTEST (A, 2) is $\begin{bmatrix} \text{false} & \text{false} \\ \text{false} & \text{true} \end{bmatrix}$ and the value of BTEST (2, A) is $\begin{bmatrix} \text{true} & \text{false} \\ \text{false} & \text{false} \end{bmatrix}$.

13.7.18 CEILING (A [, KIND])

Description. Returns the least integer greater than or equal to its argument.

Class. Elemental function.

Arguments.

A	shall be of type real.
KIND (optional)	shall be a scalar integer initialization expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default integer type.

Result Value. The result has a value equal to the least integer greater than or equal to A.

Examples. CEILING (3.7) has the value 4. CEILING (–3.7) has the value –3.

13.7.19 CHAR (I [, KIND])

Description. Returns the character in a given position of the processor collating sequence associated with the specified kind type parameter. It is the inverse of the ICHAR function.

Class. Elemental function.

Arguments.

I	shall be of type integer with a value in the range $0 \leq \text{I} \leq n-1$, where n is the number of characters in the collating sequence associated with the specified kind type parameter.
KIND (optional)	shall be a scalar integer initialization expression.

Result Characteristics. Character of length one. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default character type.

Result Value. The result is the character in position I of the collating sequence associated with the specified kind type parameter. ICHAR (CHAR (I, KIND (C))) shall have the value I for $0 \leq \text{I} \leq n-1$ and CHAR (ICHAR (C), KIND (C)) shall 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.

13.7.20 CMPLX (X [, Y, KIND])

Description. Convert to complex type.

Class. Elemental function.

Arguments.

X	shall be of type integer, real, or complex, or a *boz-literal-constant*.

Y (optional) shall be of type integer or real, or a *boz-literal-constant*. If X is of type complex, Y shall not be present, nor shall Y be associated with an optional dummy argument.

KIND (optional) shall be a scalar integer initialization expression.

Result Characteristics. The result is of type complex. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that 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 X is complex, it is as if X were real with the value REAL (X, KIND) and Y were present with the value AIMAG (X, KIND). CMPLX (X, Y, KIND) has the complex value whose real part is REAL (X, KIND) and whose imaginary part is REAL (Y, KIND).

Example. CMPLX (–3) has the value (–3.0, 0.0).

13.7.21 COMMAND_ARGUMENT_COUNT ()

Description. Returns the number of command arguments.

Class. Inquiry function.

Arguments. None.

Result Characteristics. Scalar default integer.

Result Value. The result value is equal to the number of command arguments available. If there are no command arguments available or if the processor does not support command arguments, then the result value is 0. If the processor has a concept of a command name, the command name does not count as one of the command arguments.

Example. See 13.7.42.

13.7.22 CONJG (Z)

Description. Conjugate of a complex number.

Class. Elemental function.

Argument. Z shall be of type complex.

Result Characteristics. Same as Z.

Result Value. If Z has the value (x, y), the result has the value $(x, -y)$.

Example. CONJG ((2.0, 3.0)) has the value (2.0, –3.0).

13.7.23 COS (X)

Description. Cosine function.

Class. Elemental function.

Argument. X shall be of type real or complex.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to cos(X).

If X is of type real, 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).

13.7.24 COSH (X)

Description. Hyperbolic cosine function.

Class. Elemental function.

Argument. X shall be of type real.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to cosh(X).

Example. COSH (1.0) has the value 1.5430806 (approximately).

13.7.25 COUNT (MASK [, DIM, KIND])

Description. Count the number of true elements of MASK along dimension DIM.

Class. Transformational function.

Arguments.

MASK	shall be of type logical. It shall not be scalar.
DIM (optional)	shall be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of MASK. The corresponding actual argument shall not be an optional dummy argument.
KIND (optional)	shall be a scalar integer initialization expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type. The result is scalar if DIM is absent; otherwise, the result has rank $n - 1$ and 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 a value equal to the number of true elements of MASK or has the value zero if MASK has size zero.

Case (ii): If MASK has rank one, COUNT (MASK, DIM) has a 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 /= C, DIM = 1) is [2, 0, 1] and COUNT (B /= C, DIM = 2) is [1, 2].

13.7.26 CPU_TIME (TIME)

Description. Returns the processor time.

Class. Subroutine.

Argument. TIME shall be scalar and of type real. It is an INTENT(OUT) argument that is assigned a processor-dependent approximation to the processor time in seconds. If the processor cannot return a meaningful time, a processor-dependent negative value is returned.

Example.

```
REAL T1, T2
...
CALL CPU_TIME(T1)
...   ! Code to be timed.
CALL CPU_TIME(T2)
WRITE (*,*) 'Time taken by code was ', T2-T1, ' seconds'
```

writes the processor time taken by a piece of code.

NOTE 13.7

A processor for which a single result is inadequate (for example, a parallel processor) might choose to provide an additional version for which time is an array.

The exact definition of time is left imprecise because of the variability in what different processors are able to provide. The primary purpose is to compare different algorithms on the same processor or discover which parts of a calculation are the most expensive.

The start time is left imprecise because the purpose is to time sections of code, as in the example.

Most computer systems have multiple concepts of time. One common concept is that of time expended by the processor for a given program. This might or might not include system overhead, and has no obvious connection to elapsed "wall clock" time.

13.7.27 CSHIFT (ARRAY, SHIFT [, DIM])

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 an array expression of rank two or greater. 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.

Class. Transformational function.

Arguments.

ARRAY — may be of any type. It shall not be scalar.

SHIFT — shall be of type integer and shall be scalar if ARRAY has rank one; otherwise, it shall be scalar or 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.

DIM (optional) — shall be a scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY. If DIM is omitted, it is as if it were present with the value 1.

Result Characteristics. 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, element i of the result is ARRAY (1 + MODULO (i + SHIFT – 1, SIZE (ARRAY))).

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 a value equal to CSHIFT (ARRAY (s_1, s_2, ..., $s_{\text{DIM}-1}$, :, $s_{\text{DIM}+1}$, ..., s_n), *sh*, 1), 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, SHIFT = 2) which has the value [3, 4, 5, 6, 1, 2]; CSHIFT (V, 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 $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$, the value of

CSHIFT (M, SHIFT = –1, DIM = 2) is $\begin{bmatrix} 3 & 1 & 2 \\ 6 & 4 & 5 \\ 9 & 7 & 8 \end{bmatrix}$, and the value of

CSHIFT (M, SHIFT = (/ –1, 1, 0 /), DIM = 2) is $\begin{bmatrix} 3 & 1 & 2 \\ 5 & 6 & 4 \\ 7 & 8 & 9 \end{bmatrix}$.

13.7.28 DATE_AND_TIME ([DATE, TIME, ZONE, VALUES])

Description. Returns data about the real-time clock and date in a form compatible with the representations defined in ISO 8601:1988.

Class. Subroutine.

Arguments.

DATE (optional) shall be scalar and of type default character. It is an INTENT (OUT) argument. It is assigned a value of the form *CCYYMMDD*, where *CC* is the century, *YY* is the year within the century, *MM* is the month within the year, and *DD*

VALUES (1) the year, including the century (for example, 1990), or –HUGE (0) if there is no date available;

VALUES (2) the month of the year, or –HUGE (0) if there is no date available;

VALUES (3) the day of the month, or –HUGE (0) if there is no date available;

VALUES (4) the time difference with respect to Coordinated Universal Time (UTC) in minutes, or –HUGE (0) if this information is not available;

VALUES (5) the hour of the day, in the range of 0 to 23, or –HUGE (0) if there is no clock;

VALUES (6) the minutes of the hour, in the range 0 to 59, or –HUGE (0) if there is no clock;

VALUES (7) the seconds of the minute, in the range 0 to 60, or –HUGE (0) if there is no clock;

VALUES (8) the milliseconds of the second, in the range 0 to 999, or –HUGE (0) if there is no clock.

Example.

```
INTEGER DATE_TIME (8)
CHARACTER (LEN = 10) BIG_BEN (3)
CALL DATE_AND_TIME (BIG_BEN (1), BIG_BEN (2),  &
                    BIG_BEN (3), DATE_TIME)
```

If run in Geneva, Switzerland on April 12, 1985 at 15:27:35.5 with a system configured for the local time zone, this sample would have assigned the value 19850412 to BIG_BEN (1), the value 152735.500 to BIG_BEN (2), the value +0100 to BIG_BEN (3), and the value (/ 1985, 4, 12, 60, 15, 27, 35, 500 /) to DATE_TIME.

NOTE 13.8

UTC is defined by ISO 8601:1988.

13.7.29 DBLE (A)

Description. Convert to double precision real type.

Class. Elemental function.

Argument. A shall be of type integer, real, or complex, or a *boz-literal-constant*.

Result Characteristics. Double precision real.

Result Value. The result has the value REAL (A, KIND (0.0D0)).

Example. DBLE (–3) has the value –3.0D0.

13.7.30 DIGITS (X)

Description. Returns the number of significant digits of the model representing numbers of the same type and kind type parameter as the argument.

Class. Inquiry function.

Argument. X shall be of type integer or real. It may be a scalar or an array.

Result Characteristics. Default integer scalar.

Result Value. The result has the value q if X is of type integer and p if X is of type real, where q and p are as defined in 13.4 for the model representing numbers of the same type and kind type parameter as X.

Example. DIGITS (X) has the value 24 for real X whose model is as in Note 13.4.

13.7.31 DIM (X, Y)

Description. The difference X–Y if it is positive; otherwise zero.

Class. Elemental function.

Arguments.

X	shall be of type integer or real.
Y	shall be of the same type and kind type parameter as X.

Result Characteristics. 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.

13.7.32 DOT_PRODUCT (VECTOR_A, VECTOR_B)

Description. Performs dot-product multiplication of numeric or logical vectors.

Class. Transformational function.

Arguments.

VECTOR_A	shall be of numeric type (integer, real, or complex) or of logical type. It shall be a rank-one array.
VECTOR_B	shall be of numeric type if VECTOR_A is of numeric type or of type logical if VECTOR_A is of type logical. It shall be a rank-one array. It shall be of the same size as VECTOR_A.

Result Characteristics. If the arguments are of numeric type, the type and kind type parameter of the result are those of the expression VECTOR_A * VECTOR_B determined by the types of the arguments according to 7.1.4.2. If the arguments are of type logical, the result is of type logical with the kind type parameter of the expression VECTOR_A .AND. VECTOR_B according to 7.1.4.2. The result is scalar.

Result Value.

Case (i): If VECTOR_A is of type integer or real, the result has the value SUM (VECTOR_A*VECTOR_B). If the vectors have size zero, the result has the value zero.

Case (ii): If VECTOR_A is of type complex, the result has the value SUM (CONJG (VECTOR_A)*VECTOR_B). If the vectors have size zero, the result has the value zero.

Case (iii): If VECTOR_A is of type logical, the result has the value ANY (VECTOR_A .AND. VECTOR_B). If the vectors have size zero, the result has the value false.

Example. DOT_PRODUCT ((/ 1, 2, 3 /), (/ 2, 3, 4 /)) has the value 20.

13.7.33 DPROD (X, Y)

Description. Double precision real product.

Class. Elemental function.

Arguments.

X	shall be of type default real.
Y	shall be of type default real.

Result Characteristics. Double precision real.

Result Value. The result has a value equal to a processor-dependent approximation to the product of X and Y.

Example. DPROD (–3.0, 2.0) has the value –6.0D0.

13.7.34 EOSHIFT (ARRAY, SHIFT [, BOUNDARY, DIM])

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 an array expression of rank two or greater. 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.

Class. Transformational function.

Arguments.

ARRAY — may be of any type. It shall not be scalar.

SHIFT — shall be of type integer and shall be scalar if ARRAY has rank one; otherwise, it shall be scalar or of rank $n-1$ and of shape (d_1, d_2, ..., $d_{\mathrm{DIM}-1}$, $d_{\mathrm{DIM}+1}$, ..., d_n) where (d_1, d_2, ..., d_n) is the shape of ARRAY.

BOUNDARY (optional) — shall be of the same type and type parameters as ARRAY and shall be scalar if ARRAY has rank one; otherwise, it shall be either scalar or of rank $n-1$ and of shape (d_1, d_2, ..., $d_{\mathrm{DIM}-1}$, $d_{\mathrm{DIM}+1}$, ..., d_n). BOUNDARY may be omitted for the 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
Complex	(0.0, 0.0)
Logical	false
Character (*len*)	*len* blanks

DIM (optional) — shall be scalar and of type integer with a value in the range $1 \leq \mathrm{DIM} \leq n$, where n is the rank of ARRAY. If DIM is omitted, it is as if it were present with the value 1.

Result Characteristics. The result has the type, type parameters, and shape of ARRAY.

Result Value. Element (s_1, s_2, ..., s_n) of the result has the value 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 LBOUND (ARRAY, DIM) $\leq s_{\mathrm{DIM}} + sh \leq$ UBOUND (ARRAY, 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, SHIFT = 3), which has the value [4, 5, 6, 0, 0, 0]; EOSHIFT (V, 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 $\begin{bmatrix} \mathrm{A} & \mathrm{B} & \mathrm{C} \\ \mathrm{D} & \mathrm{E} & \mathrm{F} \\ \mathrm{G} & \mathrm{H} & \mathrm{I} \end{bmatrix}$, then the value of EOSHIFT (M, SHIFT = –1, BOUNDARY = '*', DIM = 2) is $\begin{bmatrix} * & \mathrm{A} & \mathrm{B} \\ * & \mathrm{D} & \mathrm{E} \\ * & \mathrm{G} & \mathrm{H} \end{bmatrix}$, and the value of EOSHIFT (M, SHIFT = (/ –1, 1, 0 /), BOUNDARY = (/ '*', '/', '?' /), DIM = 2) is $\begin{bmatrix} * & \mathrm{A} & \mathrm{B} \\ \mathrm{E} & \mathrm{F} & / \\ \mathrm{G} & \mathrm{H} & \mathrm{I} \end{bmatrix}$.

13.7.35 EPSILON (X)

Description. Returns a positive model number that is almost negligible compared to unity of the model representing numbers of the same type and kind type parameter as the argument.

Class. Inquiry function.

Argument. X shall be of type real. It may be a scalar or an array.

Result Characteristics. Scalar of the same type and kind type parameter as X.

Result Value. The result has the value b^{1-p} where b and p are as defined in 13.4 for the model representing numbers of the same type and kind type parameter as X.

Example. EPSILON (X) has the value 2^{-23} for real X whose model is as in Note 13.4.

13.7.36 EXP (X)

Description. Exponential.

Class. Elemental function.

Argument. X shall be of type real or complex.

Result Characteristics. Same as X.

Result Value. The result has a 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).

13.7.37 EXPONENT (X)

Description. Returns the exponent part of the argument when represented as a model number.

Class. Elemental function.

Argument. X shall be of type real.

Result Characteristics. Default integer.

Result Value. The result has a value equal to the exponent e of the model representation (13.4) for the value of X, provided X is nonzero and e is within the range for default integers. If X has the value zero, the result has the value zero. If X is an IEEE infinity or NaN, the result has the value HUGE(0).

Examples. EXPONENT (1.0) has the value 1 and EXPONENT (4.1) has the value 3 for reals whose model is as in Note 13.4.

13.7.38 EXTENDS_TYPE_OF (A, MOLD)

Description. Inquires whether the dynamic type of A is an extension type (4.5.6) of the dynamic type of MOLD.

Class. Inquiry function.

Arguments.

A	shall be an object of extensible type. If it is a pointer, it shall not have an undefined association status.
MOLD	shall be an object of extensible type. If it is a pointer, it shall not have an undefined association status.

Result Characteristics. Default logical scalar.

Result Value. If MOLD is unlimited polymorphic and is either a disassociated pointer or unallocated allocatable, the result is true; otherwise if A is unlimited polymorphic and is either a disassociated pointer or unallocated allocatable, the result is false; otherwise the result is true if and only if the dynamic type of A is an extension type of the dynamic type of MOLD.

NOTE 13.9

The dynamic type of a disassociated pointer or unallocated allocatable is its declared type.

13.7.39 FLOOR (A [, KIND])

Description. Returns the greatest integer less than or equal to its argument.

Class. Elemental function.

Arguments.

A	shall be of type real.
KIND (optional)	shall be a scalar integer initialization expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default integer type.

Result Value. The result has a value equal to the greatest integer less than or equal to A.

Examples. FLOOR (3.7) has the value 3. FLOOR (–3.7) has the value –4.

13.7.40 FRACTION (X)

Description. Returns the fractional part of the model representation of the argument value.

Class. Elemental function.

Argument. X shall be of type real.

Result Characteristics. Same as X.

Result Value. The result has the value $X \times b^{-e}$, where b and e are as defined in 13.4 for the model representation of X. If X has the value zero, the result has the value zero. If X is an IEEE infinity, the result is that infinity. If X is an IEEE NaN, the result is that NaN.

Example. FRACTION (3.0) has the value 0.75 for reals whose model is as in Note 13.4.

13.7.41 GET_COMMAND ([COMMAND, LENGTH, STATUS])

Description. Returns the entire command by which the program was invoked.

Class. Subroutine.

Arguments.

COMMAND (optional)	shall be scalar and of type default character. It is an INTENT(OUT) argument. It is assigned the entire command by which the program was invoked. If the command cannot be determined, COMMAND is assigned all blanks.
LENGTH (optional)	shall be scalar and of type default integer. It is an INTENT(OUT) argument. It is assigned the significant length of the command by which the program was invoked. The significant length may include trailing blanks if the processor allows commands with significant trailing blanks. This length does not consider any possible truncation or padding in assigning the command to the COMMAND argument; in fact the COMMAND argument need not even be present. If the command length cannot be determined, a length of 0 is assigned.
STATUS (optional)	shall be scalar and of type default integer. It is an INTENT(OUT) argument. It is assigned the value -1 if the COMMAND argument is present and has a length less than the significant length of the command. It is assigned a processor-dependent positive value if the command retrieval fails. Otherwise it is assigned the value 0.

13.7.42 GET_COMMAND_ARGUMENT (NUMBER [, VALUE, LENGTH, STATUS])

Description. Returns a command argument.

Class. Subroutine.

Arguments.

NUMBER	shall be scalar and of type default integer. It is an INTENT(IN) argument.

It specifies the number of the command argument that the other arguments give information about. Useful values of NUMBER are those between 0 and the argument count returned by the COMMAND_ARGUMENT_COUNT intrinsic. Other values are allowed, but will result in error status return (see below).

Command argument 0 is defined to be the command name by which the program was invoked if the processor has such a concept. It is allowed to call the GET_COMMAND_ARGUMENT procedure for command argument number 0, even if the processor does not define command names or other command arguments.

The remaining command arguments are numbered consecutively from 1 to the argument count in an order determined by the processor.

VALUE (optional) shall be scalar and of type default character. It is an INTENT(OUT) argument. It is assigned the value of the command argument specified by NUMBER. If the command argument value cannot be determined, VALUE is assigned all blanks.

LENGTH (optional) shall be scalar and of type default integer. It is an INTENT(OUT) argument. It is assigned the significant length of the command argument specified by NUMBER. The significant length may include trailing blanks if the processor allows command arguments with significant trailing blanks. This length does not consider any possible truncation or padding in assigning the command argument value to the VALUE argument; in fact the VALUE argument need not even be present. If the command argument length cannot be determined, a length of 0 is assigned.

STATUS (optional) shall be scalar and of type default integer. It is an INTENT(OUT) argument. It is assigned the value -1 if the VALUE argument is present and has a length less than the significant length of the command argument specified by NUMBER. It is assigned a processor-dependent positive value if the argument retrieval fails. Otherwise it is assigned the value 0.

NOTE 13.10

One possible reason for failure is that NUMBER is negative or greater than COMMAND_ARGUMENT_COUNT().

Example.

```
Program echo
  integer :: i
  character :: command*32, arg*128
  call get_command_argument(0, command)
  write (*,*) "Command name is: ", command
  do i = 1 , command_argument_count()
    call get_command_argument(i, arg)
    write (*,*) "Argument ", i, " is ", arg
  end do
end program echo
```

13.7.43 GET_ENVIRONMENT_VARIABLE (NAME [, VALUE, LENGTH, STATUS, TRIM_NAME])

Description. Gets the value of an environment variable.

Class. Subroutine.

Arguments.

NAME	shall be a scalar and of type default character. It is an INTENT(IN) argument. The interpretation of case is processor dependent
VALUE (optional)	shall be a scalar of type default character. It is an INTENT(OUT) argument. It is assigned the value of the environment variable specified by NAME. VALUE is assigned all blanks if the environment variable does not exist or does not have a value or if the processor does not support environment variables.
LENGTH (optional)	shall be a scalar of type default integer. It is an INTENT(OUT) argument. If the specified environment variable exists and has a value, LENGTH is set to the length of that value. Otherwise LENGTH is set to 0.
STATUS (optional)	shall be scalar and of type default integer. It is an INTENT(OUT) argument. If the environment variable exists and either has no value or its value is successfully assigned to VALUE, STATUS is set to zero. STATUS is set to -1 if the VALUE argument is present and has a length less than the significant length of the environment variable. It is assigned the value 1 if the specified environment variable does not exist, or 2 if the processor does not support environment variables. Processor-dependent values greater than 2 may be returned for other error conditions.
TRIM_NAME (optional)	shall be a scalar of type logical. It is an INTENT(IN) argument. If TRIM_NAME is present with the value false then trailing blanks in NAME are considered significant if the processor supports trailing blanks in environment variable names. Otherwise trailing blanks in NAME are not considered part of the environment variable's name.

13.7.44 HUGE (X)

Description. Returns the largest number of the model representing numbers of the same type and kind type parameter as the argument.

Class. Inquiry function.

Argument. X shall be of type integer or real. It may be a scalar or an array.

Result Characteristics. Scalar of the same type and kind type parameter as X.

Result Value. The result has the 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 13.4 for the model representing numbers of the same type and kind type parameter as X.

Example. HUGE (X) has the value $(1 - 2^{-24}) \times 2^{127}$ for real X whose model is as in Note 13.4.

13.7.45 IACHAR (C [, KIND])

Description. Returns the position of a character in the ASCII collating sequence. This is the inverse of the ACHAR function.

Class. Elemental function.

Arguments.

C	shall be of type character and of length one.
KIND (optional)	shall be a scalar integer initialization expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default integer type.

Result Value. If C is in the collating sequence defined by the codes specified in ISO/IEC 646:1991 (International Reference Version), the result is the position of C in that sequence and satisfies the inequality ($0 \leq$ IACHAR(C) ≤ 127). A processor-dependent value is returned if C is not in the ASCII collating sequence. The results are consistent with the LGE, LGT, LLE, and LLT lexical comparison functions. For example, if LLE (C, D) is true, IACHAR (C) <= IACHAR (D) is true where C and D are any two characters representable by the processor.

Example. IACHAR ('X') has the value 88.

13.7.46 IAND (I, J)

Description. Performs a bitwise AND.

Class. Elemental function.

Arguments.

I	shall be of type integer.
J	shall be of type integer with the same kind type parameter as I.

Result Characteristics. Same as I.

Result Value. The result has the value obtained by combining I and J bit-by-bit according to the following truth table:

I	J	IAND (I, J)
1	1	1
1	0	0
0	1	0
0	0	0

The model for the interpretation of an integer value as a sequence of bits is in 13.3.

Example. IAND (1, 3) has the value 1.

13.7.47 IBCLR (I, POS)

Description. Clears one bit to zero.

Class. Elemental function.

Arguments.

I	shall be of type integer.

POS	shall be of type integer. It shall be nonnegative and less than BIT_SIZE (I).

Result Characteristics. Same as I.

Result Value. The result has the value of the sequence of bits of I, except that bit POS is zero. The model for the interpretation of an integer value as a sequence of bits is in 13.3.

Examples. IBCLR (14, 1) has the result 12. If V has the value [1, 2, 3, 4], the value of IBCLR (POS = V, I = 31) is [29, 27, 23, 15].

13.7.48 IBITS (I, POS, LEN)

Description. Extracts a sequence of bits.

Class. Elemental function.

Arguments.

I	shall be of type integer.
POS	shall be of type integer. It shall be nonnegative and POS + LEN shall be less than or equal to BIT_SIZE (I).
LEN	shall be of type integer and nonnegative.

Result Characteristics. Same as I.

Result Value. The result has the value of the sequence of LEN bits in I beginning at bit POS, right-adjusted and with all other bits zero. The model for the interpretation of an integer value as a sequence of bits is in 13.3.

Example. IBITS (14, 1, 3) has the value 7.

13.7.49 IBSET (I, POS)

Description. Sets one bit to one.

Class. Elemental function.

Arguments.

I	shall be of type integer.
POS	shall be of type integer. It shall be nonnegative and less than BIT_SIZE (I).

Result Characteristics. Same as I.

Result Value. The result has the value of the sequence of bits of I, except that bit POS is one. The model for the interpretation of an integer value as a sequence of bits is in 13.3.

Examples. IBSET (12, 1) has the value 14. If V has the value [1, 2, 3, 4], the value of IBSET (POS = V, I = 0) is [2, 4, 8, 16].

13.7.50 ICHAR (C [, KIND])

Description. Returns the position of a character in the processor collating sequence associated with the kind type parameter of the character. This is the inverse of the CHAR function.

Class. Elemental function.

Arguments.

C	shall be of type character and of length one. Its value shall be that of a character capable of representation in the processor.
KIND (optional)	shall be a scalar integer initialization expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default integer type.

Result Value. The result is the position of C in the processor collating sequence associated with the kind type parameter of C and is in the range $0 \leq \text{ICHAR(C)} \leq n - 1$, where n is the number of characters in the collating sequence. For any characters C and D capable of representation in the processor, C <= D is true if and only if ICHAR (C) <= ICHAR (D) is true and C == D is true if and only if ICHAR (C) == ICHAR (D) is true.

Example. ICHAR ('X') has the value 88 on a processor using the ASCII collating sequence for the default character type.

13.7.51 IEOR (I, J)

Description. Performs a bitwise exclusive OR.

Class. Elemental function.

Arguments.

I	shall be of type integer.
J	shall be of type integer with the same kind type parameter as I.

Result Characteristics. Same as I.

Result Value. The result has the value obtained by combining I and J bit-by-bit according to the following truth table:

I	J	IEOR (I, J)
1	1	0
1	0	1
0	1	1
0	0	0

The model for the interpretation of an integer value as a sequence of bits is in 13.3.

Example. IEOR (1, 3) has the value 2.

13.7.52 INDEX (STRING, SUBSTRING [, BACK, KIND])

Description. Returns the starting position of a substring within a string.

Class. Elemental function.

Arguments.

STRING	shall be of type character.
SUBSTRING	shall be of type character with the same kind type parameter as STRING.

BACK (optional) shall be of type logical.

KIND (optional) shall be a scalar integer initialization expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type.

Result Value.

Case (i): If BACK is absent or has the value false, the result is the minimum positive value of I such that STRING (I : I + LEN (SUBSTRING) – 1) = SUBSTRING or zero if there is no such value. Zero is returned if LEN (STRING) < LEN (SUBSTRING) and one is returned if LEN (SUBSTRING) = 0.

Case (ii): If BACK is present with the value true, the result is the maximum value of I less than or equal to LEN (STRING) – LEN (SUBSTRING) + 1 such that STRING (I : I + LEN (SUBSTRING) – 1) = SUBSTRING or zero if there is no such value. Zero is returned if LEN (STRING) < LEN (SUBSTRING) and LEN (STRING) + 1 is returned if LEN (SUBSTRING) = 0.

Examples. INDEX ('FORTRAN', 'R') has the value 3.
INDEX ('FORTRAN', 'R', BACK = .TRUE.) has the value 5.

13.7.53 INT (A [, KIND])

Description. Convert to integer type.

Class. Elemental function.

Arguments.

A shall be of type integer, real, or complex, or a *boz-literal-constant*.

KIND (optional) shall be a scalar integer initialization expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default integer type.

Result Value.

Case (i): If A is of type integer, INT (A) = A.

Case (ii): If A is of type real, 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) = INT(REAL(A, KIND(A))).

Case (iv): If A is a *boz-literal-constant*, it is treated as if it were an *int-literal-constant* with a *kind-param* that specifies the representation method with the largest decimal exponent range supported by the processor.

Example. INT (–3.7) has the value –3.

13.7.54 IOR (I, J)

Description. Performs a bitwise inclusive OR.

Class. Elemental function.

Arguments.

I	shall be of type integer.
J	shall be of type integer with the same kind type parameter as I.

Result Characteristics. Same as I.

Result Value. The result has the value obtained by combining I and J bit-by-bit according to the following truth table:

I	J	IOR (I, J)
1	1	1
1	0	1
0	1	1
0	0	0

The model for the interpretation of an integer value as a sequence of bits is in 13.3.

Example. IOR (5, 3) has the value 7.

13.7.55 ISHFT (I, SHIFT)

Description. Performs a logical shift.

Class. Elemental function.

Arguments.

I	shall be of type integer.
SHIFT	shall be of type integer. The absolute value of SHIFT shall be less than or equal to BIT_SIZE (I).

Result Characteristics. Same as I.

Result Value. The result has the value obtained by shifting the bits of I by SHIFT positions. If SHIFT is positive, the shift is to the left; if SHIFT is negative, the shift is to the right; and if SHIFT is zero, no shift is performed. Bits shifted out from the left or from the right, as appropriate, are lost. Zeros are shifted in from the opposite end. The model for the interpretation of an integer value as a sequence of bits is in 13.3.

Example. ISHFT (3, 1) has the result 6.

13.7.56 ISHFTC (I, SHIFT [, SIZE])

Description. Performs a circular shift of the rightmost bits.

Class. Elemental function.

Arguments.

I	shall be of type integer.
SHIFT	shall be of type integer. The absolute value of SHIFT shall be less than or equal to SIZE.

Result Characteristics. Same as I.

Result Value. The result has the value obtained by shifting the SIZE rightmost bits of I circularly by SHIFT positions. If SHIFT is positive, the shift is to the left; if SHIFT is negative, the shift is to the right; and if SHIFT is zero, no shift is performed. No bits are lost. The unshifted bits are unaltered. The model for the interpretation of an integer value as a sequence of bits is in 13.3.

Example. ISHFTC (3, 2, 3) has the value 5.

13.7.57 IS_IOSTAT_END (I)

Description. Determine whether a value indicates an end-of-file condition.

Class. Elemental function.

Argument. I shall be of type integer.

Result Characteristics. Default logical.

Result Value. The result has the value true if and only if I is a value for the *scalar-int-variable* in an IOSTAT= specifier (9.10.4) that would indicate an end-of-file condition.

13.7.58 IS_IOSTAT_EOR (I)

Description. Determine whether a value indicates an end-of-record condition.

Class. Elemental function.

Argument. I shall be of type integer.

Result Characteristics. Default logical.

Result Value. The result has the value true if and only if I is a value for the *scalar-int-variable* in an IOSTAT= specifier (9.10.4) that would indicate an end-of-record condition.

13.7.59 KIND (X)

Description. Returns the value of the kind type parameter of X.

Class. Inquiry function.

Argument. X may be of any intrinsic type. It may be a scalar or an array.

Result Characteristics. Default integer scalar.

Result Value. The result has a value equal to the kind type parameter value of X.

Example. KIND (0.0) has the kind type parameter value of default real.

13.7.60 LBOUND (ARRAY [, DIM, KIND])

Description. Returns all the lower bounds or a specified lower bound of an array.

Class. Inquiry function.

Arguments.

ARRAY may be of any type. It shall not be scalar. It shall not be an unallocated allocatable or a pointer that is not associated.

DIM (optional) shall be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY. The corresponding actual argument shall not be an optional dummy argument.

KIND (optional) shall be a scalar integer initialization expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type. The result 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): If ARRAY is a whole array or array structure component and either ARRAY is an assumed-size array of rank DIM or dimension DIM of ARRAY has nonzero extent, LBOUND (ARRAY, DIM) has a value equal to the lower bound for subscript DIM of ARRAY. Otherwise the result value is 1.

Case (ii): LBOUND (ARRAY) has a value whose ith component is equal to LBOUND (ARRAY, i), for $i = 1, 2, ..., n$, where n is the rank of ARRAY.

Examples. If A is declared by the statement

```
REAL A (2:3, 7:10)
```

then LBOUND (A) is [2, 7] and LBOUND (A, DIM=2) is 7.

13.7.61 LEN (STRING [, KIND])

Description. Returns the length of a character entity.

Class. Inquiry function.

Arguments.

STRING shall be of type character. It may be a scalar or an array. If it is an unallocated allocatable or a pointer that is not associated, its length type parameter shall not be deferred.

KIND (optional) shall be a scalar integer initialization expression.

Result Characteristics. Integer scalar. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type.

Result Value. The result has a value equal to the number of characters in STRING if it is scalar or in an element of STRING if it is an array.

Example. If C is declared by the statement

```
CHARACTER (11) C (100)
```

LEN (C) has the value 11.

13.7.62 LEN_TRIM (STRING [, KIND])

Description. Returns the length of the character argument without counting trailing blank characters.

Class. Elemental function.

Arguments.

STRING	shall be of type character.
KIND (optional)	shall be a scalar integer initialization expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type.

Result Value. The result has a value equal to the number of characters remaining after any trailing blanks in STRING are removed. If the argument contains no nonblank characters, the result is zero.

Examples. LEN_TRIM (' A B ') has the value 4 and LEN_TRIM (' ') has the value 0.

13.7.63 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.

Class. Elemental function.

Arguments.

STRING_A	shall be of type default character.
STRING_B	shall be of type default character.

Result Characteristics. Default 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 ASCII collating sequence; otherwise, the result is false.

NOTE 13.11

The result is true if both STRING_A and STRING_B are of zero length.

Example. LGE ('ONE', 'TWO') has the value false.

13.7.64 LGT (STRING_A, STRING_B)

Description. Test whether a string is lexically greater than another string, based on the ASCII collating sequence.

Class. Elemental function.

Arguments.

STRING_A	shall be of type default character.
STRING_B	shall be of type default character.

Result Characteristics. Default 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 ASCII collating sequence; otherwise, the result is false.

NOTE 13.12

The result is false if both STRING_A and STRING_B are of zero length.

Example. LGT ('ONE', 'TWO') has the value false.

13.7.65 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.

Class. Elemental function.

Arguments.

STRING_A	shall be of type default character.
STRING_B	shall be of type default character.

Result Characteristics. Default 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 ASCII collating sequence; otherwise, the result is false.

NOTE 13.13

The result is true if both STRING_A and STRING_B are of zero length.

Example. LLE ('ONE', 'TWO') has the value true.

13.7.66 LLT (STRING_A, STRING_B)

Description. Test whether a string is lexically less than another string, based on the ASCII collating sequence.

Class. Elemental function.

Arguments.

STRING_A	shall be of type default character.
STRING_B	shall be of type default character.

Result Characteristics. Default 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 ASCII collating sequence; otherwise, the result is false.

NOTE 13.14

The result is false if both STRING_A and STRING_B are of zero length.

Example. LLT ('ONE', 'TWO') has the value true.

13.7.67 LOG (X)

Description. Natural logarithm.

Class. Elemental function.

Argument. X shall be of type real or complex. If X is real, its value shall be greater than zero. If X is complex, its value shall not be zero.

Result Characteristics. Same as X.

Result Value. The result has a 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 $-\pi \leq \omega \leq \pi$. If the real part of X is less than zero and the imaginary part of X is zero, then the imaginary part of the result is π if the imaginary part of X is positive real zero or the processor cannot distinguish between positive and negative real zero, and $-\pi$ if the imaginary part of X is negative real zero.

Example. LOG (10.0) has the value 2.3025851 (approximately).

13.7.68 LOG10 (X)

Description. Common logarithm.

Class. Elemental function.

Argument. X shall be of type real. The value of X shall be greater than zero.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to $\log_{10} X$.

Example. LOG10 (10.0) has the value 1.0 (approximately).

13.7.69 LOGICAL (L [, KIND])

Description. Converts between kinds of logical.

Class. Elemental function.

Arguments.

L	shall be of type logical.
KIND (optional)	shall be a scalar integer initialization expression.

Result Characteristics. Logical. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default logical.

Result Value. The value is that of L.

Example. LOGICAL (L .OR. .NOT. L) has the value true and is of type default logical, regardless of the kind type parameter of the logical variable L.

13.7.70 MATMUL (MATRIX_A, MATRIX_B)

Description. Performs matrix multiplication of numeric or logical matrices.

Class. Transformational function.

Arguments.

MATRIX_A	shall be of numeric type (integer, real, or complex) or of logical type. It shall be an array of rank one or two.
MATRIX_B	shall be of numeric type if MATRIX_A is of numeric type and of logical type if MATRIX_A is of logical type. It shall be an array of rank one or two. If MATRIX_A has rank one, MATRIX_B shall have rank two. If MATRIX_B has rank one, MATRIX_A shall have rank two. The size of the first (or only) dimension of MATRIX_B shall equal the size of the last (or only) dimension of MATRIX_A.

Result Characteristics. If the arguments are of numeric type, the type and kind type parameter of the result are determined by the types of the arguments as specified in 7.1.4.2 for the * operator. If the arguments are of type logical, the result is of type logical with the kind type parameter of the arguments as specified in 7.1.4.2 for the .AND. operator. 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 the value SUM (MATRIX_A $(i, :)$ * MATRIX_B $(:, j)$) if the arguments are of numeric type and has the value ANY (MATRIX_A $(i, :)$ .AND. MATRIX_B $(:, j)$) if the arguments are of logical type.

Case (ii): Element (j) of the result has the value SUM (MATRIX_A $(:)$ * MATRIX_B $(:, j)$) if the arguments are of numeric type and has the value ANY (MATRIX_A $(:)$ .AND. MATRIX_B $(:, j)$) if the arguments are of logical type.

Case (iii): Element (i) of the result has the value SUM (MATRIX_A $(i, :)$ * MATRIX_B $(:)$) if the arguments are of numeric type and has the 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 the value $\begin{bmatrix} 14 & 20 \\ 20 & 29 \end{bmatrix}$.

Case (ii): The result of MATMUL (X, A) is the vector-matrix product XA with the value [5, 8, 11].

Case (iii): The result of MATMUL (A, Y) is the matrix-vector product AY with the value [14, 20].

13.7.71 MAX (A1, A2 [, A3, ...])

Description. Maximum value.

Class. Elemental function.

Arguments. The arguments shall all have the same type which shall be integer, real, or character and they shall all have the same kind type parameter.

Result Characteristics. The type and kind type parameter of the result are the same as those of the arguments. For arguments of character type, the length of the result is the length of the longest argument.

Result Value. The value of the result is that of the largest argument. For arguments of character type, the result is the value that would be selected by application of intrinsic relational operators; that is, the collating sequence for characters with the kind type parameter of the arguments is applied. If the selected argument is shorter than the longest argument, the result is extended with blanks on the right to the length of the longest argument.

Examples. MAX (–9.0, 7.0, 2.0) has the value 7.0, MAX ("Z", "BB") has the value "Z ", and MAX ((/"A", "Z"/), (/"BB", "Y "/)) has the value (/"BB", "Z "/).

13.7.72 MAXEXPONENT (X)

Description. Returns the maximum exponent of the model representing numbers of the same type and kind type parameter as the argument.

Class. Inquiry function.

Argument. X shall be of type real. It may be a scalar or an array.

Result Characteristics. Default integer scalar.

Result Value. The result has the value $e_{\max}$, as defined in 13.4 for the model representing numbers of the same type and kind type parameter as X.

Example. MAXEXPONENT (X) has the value 127 for real X whose model is as in Note 13.4.

13.7.73 MAXLOC (ARRAY, DIM [, MASK, KIND]) or MAXLOC (ARRAY [, MASK, KIND])

Description. Determine the location of the first element of ARRAY along dimension DIM having the maximum value of the elements identified by MASK.

Class. Transformational function.

Arguments.

ARRAY	shall be of type integer, real, or character. It shall not be scalar.
DIM	shall be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY. The corresponding actual argument shall not be an optional dummy argument.
MASK (optional)	shall be of type logical and shall be conformable with ARRAY.

KIND (optional) shall be a scalar integer initialization expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type. If DIM is absent, the result is an array of rank one and of size equal to the rank of ARRAY; otherwise, the result is of rank $n-1$ and shape $(d_1, d_2, ..., d_{\mathrm{DIM}-1}, d_{\mathrm{DIM}+1}, ..., d_n)$, where $(d_1, d_2, ..., d_n)$ is the shape of ARRAY.

Result Value.

Case (i): The result of MAXLOC (ARRAY) is a rank-one array whose element values are the values of the subscripts of an element of ARRAY whose value equals the maximum value of all of the elements of ARRAY. The ith subscript returned lies in the range 1 to e_i, where e_i is the extent of the ith dimension of ARRAY. If more than one element has the maximum value, the element whose subscripts are returned is the first such element, taken in array element order. If ARRAY has size zero, all elements of the result are zero.

Case (ii): The result of MAXLOC (ARRAY, MASK = MASK) is a rank-one array whose element values are the values of the subscripts 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 ith subscript returned lies in the range 1 to e_i, where e_i is the extent of the ith dimension of ARRAY. If more than one such element has the maximum value, the element whose subscripts are returned is the first such element taken in array element order. If ARRAY has size zero or every element of MASK has the value false, all elements of the result are zero.

Case (iii): If ARRAY has rank one, MAXLOC (ARRAY, DIM = DIM [, MASK = MASK]) is a scalar whose value is equal to that of the first element of MAXLOC (ARRAY [, MASK = MASK]). Otherwise, the value of element $(s_1, s_2, ..., s_{\mathrm{DIM}-1}, s_{\mathrm{DIM}+1}, ..., s_n)$ of the result is equal to

MAXLOC (ARRAY $(s_1, s_2, ..., s_{\mathrm{DIM}-1}, :, s_{\mathrm{DIM}+1}, ..., s_n)$, DIM=1
[, MASK = MASK $(s_1, s_2, ..., s_{\mathrm{DIM}-1}, :, s_{\mathrm{DIM}+1}, ..., s_n)$]).

If ARRAY has type character, the result is the value that would be selected by application of intrinsic relational operators; that is, the collating sequence for characters with the kind type parameter of the arguments is applied.

Examples.

Case (i): The value of MAXLOC ((/ 2, 6, 4, 6 /)) is [2].

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 ¡ 6) has the value [3, 2]. Note that this is independent of the declared lower bounds for A.

Case (iii): The value of MAXLOC ((/ 5, -9, 3 /), DIM = 1) is 1. If B has the value

[1 3 −9

1i2Td[(1)6Td[(4)]TJ/F19.947.5926TJ/231.49525Td[(n)]TJ/F89.96325801047-2.092Td[20Td[(MA

Arguments.

ARRAY	shall be of type integer, real, or character. It shall not be scalar.
DIM	shall be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY. The corresponding actual argument shall not be an optional dummy argument.
MASK (optional)	shall be of type logical and shall be conformable with ARRAY.

Result Characteristics. The result is of the same type and type parameters as ARRAY. It is scalar if DIM is absent; otherwise, the result has rank $n-1$ and 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 a value equal to the maximum value of all the elements of ARRAY if the size of ARRAY is not zero. If ARRAY has size zero and type integer or real, the result has the value of the negative number of the largest magnitude supported by the processor for numbers of the type and kind type parameter of ARRAY. If ARRAY has size zero and type character, the result has the value of a string of characters of length LEN (ARRAY), with each character equal to CHAR (0, KIND = KIND (ARRAY)).

Case (ii): The result of MAXVAL (ARRAY, MASK = MASK) has a value equal to that of MAXVAL (PACK (ARRAY, MASK)).

Case (iii): The result of MAXVAL (ARRAY, DIM = DIM [,MASK = MASK]) has a value equal to that of MAXVAL (ARRAY [,MASK = MASK]) if ARRAY has rank one. Otherwise, the value of element (s_1, s_2, ..., $s_{\text{DIM}-1}$, $s_{\text{DIM}+1}$, ..., s_n) of the result is equal to

MAXVAL (ARRAY (s_1, s_2, ..., $s_{\text{DIM}-1}$, :, $s_{\text{DIM}+1}$, ..., s_n)
[, MASK = MASK (s_1, s_2, ..., $s_{\text{DIM}-1}$, :, $s_{\text{DIM}+1}$, ..., s_n)]).

If ARRAY has type character, the result is the value that would be selected by application of intrinsic relational operators; that is, the collating sequence for characters with the kind type parameter of the arguments is applied.

Examples.

Case (i): The value of MAXVAL ((/ 1, 2, 3 /)) is 3.

Case (ii): MAXVAL (C, MASK = C $<$ 0.0) finds the maximum of the negative 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].

13.7.75 MERGE (TSOURCE, FSOURCE, MASK)

Description. Choose alternative value according to the value of a mask.

Class. Elemental function.

Arguments.

TSOURCE	may be of any type.
FSOURCE	shall be of the same type and type parameters as TSOURCE.

MASK shall be of type logical.

Result Characteristics. Same as TSOURCE.

Result Value. The result is TSOURCE if MASK is true and FSOURCE otherwise.

Examples. 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 $\begin{bmatrix} 1 & 3 & 5 \\ 7 & 4 & 6 \end{bmatrix}$. The value of MERGE (1.0, 0.0, K > 0) is 1.0 for K = 5 and 0.0 for K = –2.

13.7.76 MIN (A1, A2 [, A3, ...])

Description. Minimum value.

Class. Elemental function.

Arguments. The arguments shall all be of the same type which shall be integer, real, or character and they shall all have the same kind type parameter.

Result Characteristics. The type and kind type parameter of the result are the same as those of the arguments. For arguments of character type, the length of the result is the length of the longest argument.

Result Value. The value of the result is that of the smallest argument. For arguments of character type, the result is the value that would be selected by application of intrinsic relational operators; that is, the collating sequence for characters with the kind type parameter of the arguments is applied. If the selected argument is shorter than the longest argument, the result is extended with blanks on the right to the length of the longest argument.

Example. MIN (–9.0, 7.0, 2.0) has the value –9.0, MIN (”A”, ”YY”) has the value “A “, and MIN ((/”Z”, ”A”/), (/”YY”, ”B “/)) has the value (/”YY”, ”A “/).

13.7.77 MINEXPONENT (X)

Description. Returns the minimum (most negative) exponent of the model representing numbers of the same type and kind type parameter as the argument.

Class. Inquiry function.

Argument. X shall be of type real. It may be a scalar or an array.

Result Characteristics. Default integer scalar.

Result Value. The result has the value $e_{\min}$, as defined in 13.4 for the model representing numbers of the same type and kind type parameter as X.

Example. MINEXPONENT (X) has the value –126 for real X whose model is as in Note 13.4.

13.7.78 MINLOC (ARRAY, DIM [, MASK, KIND]) or MINLOC (ARRAY [, MASK, KIND])

Description. Determine the location of the first element of ARRAY along dimension DIM having the minimum value of the elements identified by MASK.

Class. Transformational function.

Arguments.

ARRAY	shall be of type integer, real, or character. It shall not be scalar.
DIM	shall be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY. The corresponding actual argument shall not be an optional dummy argument.
MASK (optional)	shall be of type logical and shall be conformable with ARRAY.
KIND (optional)	shall be a scalar integer initialization expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type. If DIM is absent, the result is an array of rank one and of size equal to the rank of ARRAY; otherwise, the result is of rank $n - 1$ and 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 MINLOC (ARRAY) is a rank-one array whose element values are the values of the subscripts of an element of ARRAY whose value equals the minimum value of all the elements of ARRAY. The ith subscript returned lies in the range 1 to e_i, where e_i is the extent of the ith dimension of ARRAY. If more than one element has the minimum value, the element whose subscripts are returned is the first such element, taken in array element order. If ARRAY has size zero, all elements of the result are zero.

Case (ii): The result of MINLOC (ARRAY, MASK = MASK) is a rank-one array whose element values are the values of the subscripts 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 ith subscript returned lies in the range 1 to e_i, where e_i is the extent of the ith dimension of ARRAY. If more than one such element has the minimum value, the element whose subscripts are returned is the first such element taken in array element order. If ARRAY has size zero or every element of MASK has the value false, all elements of the result are zero.

Case (iii): If ARRAY has rank one, MINLOC (ARRAY, DIM = DIM [, MASK = MASK]) is a scalar whose value is equal to that of the first element of MINLOC (ARRAY [, MASK = MASK]). Otherwise, the value of element $(s_1, s_2, ..., s_{\text{DIM}-1}, s_{\text{DIM}+1}, ..., s_n)$ of the result is equal to

MINLOC (ARRAY $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$, DIM=1
[, MASK = MASK $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$]).

If ARRAY has type character, the result is the value that would be selected by application of intrinsic relational operators; that is, the collating sequence for characters with the kind type parameter of the arguments is applied.

Examples.

Case (i): The value of MINLOC ((/ 4, 3, 6, 3 /)) is [2].

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 > –4) has the value [1, 4]. Note that this is true even if A has a declared lower bound other than 1.

Case (iii): The value of MINLOC ((/ 5, -9, 3 /), DIM = 1) is 2. If B has the value $\begin{bmatrix} 1 & 3 & -9 \\ 2 & 2 & 6 \end{bmatrix}$, MINLOC (B, DIM = 1) is [1, 2, 1] and MINLOC (B, DIM = 2) is [3, 1]. Note that this is true even if B has a declared lower bound other than 1.

13.7.79 MINVAL (ARRAY, DIM [, MASK]) or MINVAL (ARRAY [, MASK])

Description. Minimum value of all the elements of ARRAY along dimension DIM corresponding to true elements of MASK.

Class. Transformational function.

Arguments.

ARRAY shall be of type integer, real, or character. It shall not be scalar.

DIM shall be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY. The corresponding actual argument shall not be an optional dummy argument.

MASK (optional) shall be of type logical and shall be conformable with ARRAY.

Result Characteristics. The result is of the same type and type parameters as ARRAY. It is scalar if DIM is absent; otherwise, the result has rank $n-1$ and 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 a value equal to the minimum value of all the elements of ARRAY if the size of ARRAY is not zero. If ARRAY has size zero and type integer or real, the result has the value of the positive number of the largest magnitude supported by the processor for numbers of the type and kind type parameter of ARRAY. If ARRAY has size zero and type character, the result has the value of a string of characters of length LEN (ARRAY), with each character equal to CHAR ($n-1$, KIND = KIND (ARRAY)), where n is the number of characters in the collating sequence for characters with the kind type parameter of ARRAY.

Case (ii): The result of MINVAL (ARRAY, MASK = MASK) has a value equal to that of MINVAL (PACK (ARRAY, MASK)).

Case (iii): The result of MINVAL (ARRAY, DIM = DIM [, MASK = MASK]) has a value equal to that of MINVAL (ARRAY [, MASK = MASK]) if ARRAY has rank one. Otherwise, the value of element $(s_1, s_2, ..., s_{\text{DIM}-1}, s_{\text{DIM}+1}, ..., s_n)$ of the result is equal to

MINVAL (ARRAY $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$
[, MASK= MASK $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$]).

If ARRAY has type character, the result is the value that would be selected by application of intrinsic relational operators; that is, the collating sequence for characters with the kind type parameter of the arguments is applied.

Examples.

Case (i): The value of MINVAL ((/ 1, 2, 3 /)) is 1.

Case (ii): MINVAL (C, MASK = C > 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].

13.7.80 MOD (A, P)

Description. Remainder function.

Class. Elemental function.

Arguments.

A shall be of type integer or real.

P shall be of the same type and kind type parameter as A. P shall not be zero.

Result Characteristics. Same as A.

Result Value. The value of the result is A–INT (A/P) * P.

Examples. MOD (3.0, 2.0) has the value 1.0 (approximately). MOD (8, 5) has the value 3. MOD (–8, 5) has the value –3. MOD (8, –5) has the value 3. MOD (–8, –5) has the value –3.

13.7.81 MODULO (A, P)

Description. Modulo function.

Class. Elemental function.

Arguments.

A shall be of type integer or real.

P shall be of the same type and kind type parameter as A. P shall not be zero.

Result Characteristics. Same as A.

Result Value.

Case (i): A is of type integer. MODULO (A, P) has the value R such that $A = Q \times P + R$, where Q is an integer, the inequalities $0 \leq R < P$ hold if $P > 0$, and $P < R \leq 0$ hold if $P < 0$.

Case (ii): A is of type real. The value of the result is A – FLOOR (A / P) * P.

Examples. MODULO (8, 5) has the value 3. MODULO (–8, 5) has the value 2. MODULO (8, –5) has the value –2. MODULO (–8, –5) has the value –3.

13.7.82 MOVE_ALLOC (FROM, TO)

Description. Moves an allocation from one allocatable object to another.

Class. Subroutine.

Arguments.

FROM may be of any type and rank. It shall be allocatable. It is an INTENT(INOUT) argument.

TO shall be type compatible (5.1.1.2) with FROM and have the same rank. It shall be allocatable. It shall be polymorphic if FROM is polymorphic. It is an INTENT(OUT) argument. Each nondeferred parameter of the declared type of TO shall have the same value as the corresponding parameter of the declared type of FROM.

The allocation status of TO becomes unallocated if FROM is unallocated on entry to MOVE_ALLOC. Otherwise, TO becomes allocated with dynamic type, type parameters, array bounds, and value identical to those that FROM had on entry to MOVE_ALLOC.

If TO has the TARGET attribute, any pointer associated with FROM on entry to MOVE_ALLOC becomes correspondingly associated with TO. If TO does not have the TARGET attribute, the pointer association status of any pointer associated with FROM on entry becomes undefined.

The allocation status of FROM becomes unallocated.

Example.

```
REAL,ALLOCATABLE :: GRID(:),TEMPGRID(:)
...
ALLOCATE(GRID(-N:N))       ! initial allocation of GRID
...
! "reallocation" of GRID to allow intermediate points
ALLOCATE(TEMPGRID(-2*N:2*N)) ! allocate bigger grid
TEMPGRID(::2)=GRID ! distribute values to new locations
CALL MOVE_ALLOC(TO=GRID,FROM=TEMPGRID)
                  ! old grid is deallocated because TO is
                  ! INTENT(OUT), and GRID then "takes over"
                  ! new grid allocation
```

NOTE 13.15

It is expected that the implementation of allocatable objects will typically involve descriptors to locate the allocated storage; MOVE_ALLOC could then be implemented by transferring the contents of the descriptor for FROM to the descriptor for TO and clearing the descriptor for FROM.

13.7.83 MVBITS (FROM, FROMPOS, LEN, TO, TOPOS)

Description. Copies a sequence of bits from one data object to another.

Class. Elemental subroutine.

Arguments.

FROM shall be of type integer. It is an INTENT (IN) argument.

FROMPOS shall be of type integer and nonnegative. It is an INTENT (IN) argument. FROMPOS + LEN shall be less than or equal to BIT_SIZE (FROM). The model for the interpretation of an integer value as a sequence of bits is in 13.3.

LEN shall be of type integer and nonnegative. It is an INTENT (IN) argument.

TO	shall be a variable of type integer with the same kind type parameter value as FROM and may be associated with FROM (12.7.3). It is an INTENT (INOUT) argument. TO is defined by copying the sequence of bits of length LEN, starting at position FROMPOS of FROM to position TOPOS of TO. No other bits of TO are altered. On return, the LEN bits of TO starting at TOPOS are equal to the value that the LEN bits of FROM starting at FROMPOS had on entry. The model for the interpretation of an integer value as a sequence of bits is in 13.3.
TOPOS	shall be of type integer and nonnegative. It is an INTENT (IN) argument. TOPOS + LEN shall be less than or equal to BIT_SIZE (TO).

Example. If TO has the initial value 6, the value of TO after the statement CALL MVBITS (7, 2, 2, TO, 0) is 5.

13.7.84 NEAREST (X, S)

Description. Returns the nearest different machine-representable number in a given direction.

Class. Elemental function.

Arguments.

X	shall be of type real.
S	shall be of type real and not equal to zero.

Result Characteristics. Same as X.

Result Value. The result has a 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 in Note 13.4.

NOTE 13.16

Unlike other floating-point manipulation functions, NEAREST operates on machine-representable numbers rather than model numbers. On many systems there are machine-representable numbers that lie between adjacent model numbers.

13.7.85 NEW_LINE (A)

Description. Returns a newline character.

Class. Inquiry function.

Argument. A shall be of type character. It may be a scalar or an array.

Result Characteristics. Character scalar of length one with the same kind type parameter as A.

Result Value.

Case (i): If A is of the default character type and the character in position 10 of the ASCII collating sequence is representable in the default character set, then the result is ACHAR(10).

Case (ii): If A is of the ASCII character type or the ISO 10646 character type, then the result is CHAR(10,KIND(A)).

Case (iii): Otherwise, the result is a processor-dependent character that represents a newline in output to files connected for formatted stream output if there is such a character.

Case (iv): Otherwise, the result is the blank character.

13.7.86 NINT (A [, KIND])

Description. Nearest integer.

Class. Elemental function.

Arguments.

A	shall be of type real.
KIND (optional)	shall be a scalar integer initialization expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default integer type.

Result Value. The result is the integer nearest A, or if there are two integers equally near A, the result is whichever such integer has the greater magnitude.

Example. NINT (2.783) has the value 3.

13.7.87 NOT (I)

Description. Performs a bitwise complement.

Class. Elemental function.

Argument. I shall be of type integer.

Result Characteristics. Same as I.

Result Value. The result has the value obtained by complementing I bit-by-bit according to the following truth table:

I	NOT (I)
1	0
0	1

The model for the interpretation of an integer value as a sequence of bits is in 13.3.

Example. If I is represented by the string of bits 01010101, NOT (I) has the binary value 10101010.

13.7.88 NULL ([MOLD])

Description. Returns a disassociated pointer or designates an unallocated allocatable component of a structure constructor.

Class. Transformational function.

Argument. MOLD shall be a pointer or allocatable. It may be of any type or may be a procedure pointer. If MOLD is a pointer its pointer association status may be undefined, disassociated, or associated. If MOLD is allocatable its allocation status may be allocated or unallocated. It need not be defined with a value.

Result Characteristics. If MOLD is present, the characteristics are the same as MOLD. If MOLD has deferred type parameters, those type parameters of the result are deferred.

If MOLD is absent, the characteristics of the result are determined by the entity with which the reference is associated. See Table 13.1. MOLD shall not be absent in any other context. If any type parameters of the contextual entity are deferred, those type parameters of the result are deferred. If any type parameters of the contextual entity are assumed, MOLD shall be present.

If the context of the reference to NULL is an actual argument to a generic procedure, MOLD shall be present if the type, type parameters, or rank is required to resolve the generic reference. MOLD shall also be present if the reference appears as an actual argument corresponding to a dummy argument with assumed character length.

Table 13.1: **Characteristics of the result of NULL ( )**

Appearance of NULL ( )	Type, type parameters, and rank of result:
right side of a pointer assignment	pointer on the left side
initialization for an object in a declaration	the object
default initialization for a component	the component
in a structure constructor	the corresponding component
as an actual argument	the corresponding dummy argument
in a DATA statement	the corresponding pointer object

Result. The result is a disassociated pointer or an unallocated allocatable entity.

Examples.

Case (i): REAL, POINTER, DIMENSION(:) :: VEC => NULL ( ) defines the initial association status of VEC to be disassociated.

Case (ii): The MOLD argument is required in the following:

```
INTERFACE GEN
    SUBROUTINE S1 (J, PI)
        INTEGER J
        INTEGER, POINTER :: PI
    END SUBROUTINE S1
    SUBROUTINE S2 (K, PR)
        INTEGER K
        REAL, POINTER :: PR
    END SUBROUTINE S2
END INTERFACE
REAL, POINTER :: REAL_PTR
CALL GEN (7, NULL (REAL_PTR) )      ! Invokes S2
```

13.7.89 PACK (ARRAY, MASK [, VECTOR])

Description. Pack an array into an array of rank one under the control of a mask.

Class. Transformational function.

Arguments.

ARRAY	may be of any type. It shall not be scalar.
MASK	shall be of type logical and shall be conformable with ARRAY.
VECTOR (optional)	shall be of the same type and type parameters as ARRAY and shall have rank one. VECTOR shall have at least as many elements as there are true elements in MASK. If MASK is scalar with the value true, VECTOR shall have at least as many elements as there are in ARRAY.

Result Characteristics. 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 the value true, in which case the result size is the size of ARRAY.

Result Value. Element i of the result is the element of ARRAY that corresponds to the ith true element of MASK, taking elements in array element order, for $i = 1, 2, ..., t$. If VECTOR is present and has size $n > t$, element i of the result has the value VECTOR (i), for $i = t + 1, ..., n$.

Examples. The nonzero elements of an array M with the 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 /= 0) is [9, 7] and the result of PACK (M, M /= 0, VECTOR = (/ 2, 4, 6, 8, 10, 12 /)) is [9, 7, 6, 8, 10, 12].

13.7.90 PRECISION (X)

Description. Returns the decimal precision of the model representing real numbers with the same kind type parameter as the argument.

Class. Inquiry function.

Argument. X shall be of type real or complex. It may be a scalar or an array.

Result Characteristics. Default integer scalar.

Result Value. The result has the value INT $((p - 1)$ * LOG10 $(b)) + k$, where b and p are as defined in 13.4 for the model representing real numbers with the same value for the kind type parameter as X, and where k is 1 if b is an integral power of 10 and 0 otherwise.

Example. PRECISION (X) has the value INT (23 * LOG10 (2.)) = INT (6.92...) = 6 for real X whose model is as in Note 13.4.

13.7.91 PRESENT (A)

Description. Determine whether an optional argument is present.

Class. Inquiry function.

Argument. A shall be the name of an optional dummy argument that is accessible in the subprogram in which the PRESENT function reference appears. It may be of any type and it may be a pointer. It may be a scalar or an array. It may be a dummy procedure. The dummy argument A has no INTENT attribute.

Result Characteristics. Default logical scalar.

Result Value. The result has the value true if A is present (12.4.1.6) and otherwise has the value false.

13.7.92 PRODUCT (ARRAY, DIM [, MASK]) or PRODUCT (ARRAY [, MASK])

Description. Product of all the elements of ARRAY along dimension DIM corresponding to the true elements of MASK.

Class. Transformational function.

Arguments.

ARRAY	shall be of type integer, real, or complex. It shall not be scalar.
DIM	shall be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY. The corresponding actual argument shall not be an optional dummy argument.
MASK (optional)	shall be of type logical and shall be conformable with ARRAY.

Result Characteristics. The result is of the same type and kind type parameter as ARRAY. It is scalar if DIM is absent; otherwise, the result has rank $n - 1$ and 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 a value equal to a processor-dependent approximation to the product of all the elements of ARRAY or has the value one if ARRAY has size zero.

Case (ii): The result of PRODUCT (ARRAY, MASK = MASK) has a value equal to a processor-dependent approximation to the product of the elements of ARRAY corresponding to the true elements of MASK or has the value one if there are no true elements.

Case (iii): If ARRAY has rank one, PRODUCT (ARRAY, DIM = DIM [, MASK = MASK]) has a value equal to that of PRODUCT (ARRAY [, MASK = MASK]). Otherwise, the value of element (s_1, s_2, ..., $s_{\text{DIM}-1}$, $s_{\text{DIM}+1}$, ..., s_n) of PRODUCT (ARRAY, DIM = DIM [,MASK = MASK]) is equal to

PRODUCT (ARRAY (s_1, s_2, ..., $s_{\text{DIM}-1}$, :, $s_{\text{DIM}+1}$, ..., s_n) [, MASK = 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 > 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].

13.7.93 RADIX (X)

Description. Returns the base of the model representing numbers of the same type and kind type parameter as the argument.

Class. Inquiry function.

Argument. X shall be of type integer or real. It may be a scalar or an array.

Result Characteristics. Default integer scalar.

Result Value. The result has the value r if X is of type integer and the value b if X is of type real, where r and b are as defined in 13.4 for the model representing numbers of the same type and kind type parameter as X.

Example. RADIX (X) has the value 2 for real X whose model is as in Note 13.4.

13.7.94 RANDOM_NUMBER (HARVEST)

Description. Returns one pseudorandom number or an array of pseudorandom numbers from the uniform distribution over the range $0 \leq x < 1$.

Class. Subroutine.

Argument. HARVEST shall be of type real. It is an INTENT (OUT) argument. It may be a scalar or an array variable. It is assigned pseudorandom numbers from the uniform distribution in the interval $0 \leq x < 1$.

Examples.

```
REAL X, Y (10, 10)
! Initialize X with a pseudorandom number
CALL RANDOM_NUMBER (HARVEST = X)
CALL RANDOM_NUMBER (Y)
! X and Y contain uniformly distributed random numbers
```

13.7.95 RANDOM_SEED ([SIZE, PUT, GET])

Description. Restarts or queries the pseudorandom number generator used by RANDOM_NUMBER.

Class. Subroutine.

Arguments. There shall either be exactly one or no arguments present.

SIZE (optional)	shall be scalar and of type default integer. It is an INTENT (OUT) argument. It is assigned the number N of integers that the processor uses to hold the value of the seed.
PUT (optional)	shall be a default integer array of rank one and size $\geq N$. It is an INTENT (IN) argument. It is used in a processor-dependent manner to compute the seed value accessed by the pseudorandom number generator.
GET (optional)	shall be a default integer array of rank one and size $\geq N$ It is an INTENT (OUT) argument. It is assigned the current value of the seed.

If no argument is present, the processor assigns a processor-dependent value to the seed.

The pseudorandom number generator used by RANDOM_NUMBER maintains a seed that is updated during the execution of RANDOM_NUMBER and that may be specified or returned by RANDOM_SEED. Computation of the seed from the argument PUT is performed in a processor-dependent manner. The value returned by GET need not be the same as the value specified by PUT in an immediately preceding reference to RANDOM_SEED. For example, following execution of the statements

```
  CALL RANDOM_SEED (PUT=SEED1)
  CALL RANDOM_SEED (GET=SEED2)
```

SEED2 need not equal SEED1. When the values differ, the use of either value as the PUT argument in a subsequent call to RANDOM_SEED shall result in the same sequence of pseudorandom numbers being generated. For example, after execution of the statements

```
  CALL RANDOM_SEED (PUT=SEED1)
  CALL RANDOM_SEED (GET=SEED2)
  CALL RANDOM_NUMBER (X1)
  CALL RANDOM_SEED (PUT=SEED2)
  CALL RANDOM_NUMBER (X2)
```

X2 equals X1.

Examples.

```
  CALL RANDOM_SEED                      ! Processor initialization
  CALL RANDOM_SEED (SIZE = K)           ! Puts size of seed in K
  CALL RANDOM_SEED (PUT = SEED (1 : K)) ! Define seed
  CALL RANDOM_SEED (GET = OLD (1 : K))  ! Read current seed
```

13.7.96 RANGE (X)

Description. Returns the decimal exponent range of the model representing integer or real numbers with the same kind type parameter as the argument.

Class. Inquiry function.

Argument. X shall be of type integer, real, or complex. It may be a scalar or an array.

Result Characteristics. Default integer scalar.

Result Value.

Case (i): For an integer argument, the result has the value INT (LOG10 (HUGE(X))).

Case (ii): For a real argument, the result has the value INT (MIN (LOG10 (HUGE(X)), –LOG10 (TINY(X)))).

Case (iii): For a complex argument, the result has the value RANGE(REAL(X)).

Examples. RANGE (X) has the value 38 for real X whose model is as in Note 13.4, because in this case HUGE(X) $= (1 - 2^{-24}) \times 2^{127}$ and TINY(X) $= 2^{-127}$.

13.7.97 REAL (A [, KIND])

Description. Convert to real type.

Class. Elemental function.

Arguments.

A shall be of type integer, real, or complex, or a *boz-literal-constant*.

KIND (optional) shall be a scalar integer initialization expression.

Result Characteristics. Real.

Case (i): If A is of type integer or real and KIND is present, the kind type parameter is that specified by the value of KIND. If A is of type integer or real and KIND is not present, the kind type parameter is that of default real type.

Case (ii): If A is of type complex and KIND is present, the kind type parameter is that specified by the value of KIND. If A is of type complex and KIND is not present, the kind type parameter is the kind type parameter of A.

Case (iii): If A is a *boz-literal-constant* and KIND is present, the kind type parameter is that specified by the value of KIND. If A is a *boz-literal-constant* and KIND is not present, the kind type parameter is that of default real type.

Result Value.

Case (i): If A is of type integer or real, the result is equal to a processor-dependent approximation to A.

Case (ii): If A is of type complex, the result is equal to a processor-dependent approximation to the real part of A.

Case (iii): If A is a *boz-literal-constant*, the value of the result is equal to the value that a variable of the same type and kind type parameters as the result would have if its value were the bit pattern specified by the *boz-literal-constant*. The interpretation of the value of the bit pattern is processor dependent.

Examples. REAL (–3) has the value –3.0. REAL (Z) has the same kind type parameter and the same value as the real part of the complex variable Z.

13.7.98 REPEAT (STRING, NCOPIES)

Description. Concatenate several copies of a string.

Class. Transformational function.

Arguments.

STRING shall be scalar and of type character.

NCOPIES shall be scalar and of type integer. Its value shall not be negative.

Result Characteristics. Character scalar of length NCOPIES times that of STRING, with the same kind type parameter as STRING.

Result Value. The value of the result is the concatenation of NCOPIES copies of STRING.

Examples. REPEAT ('H', 2) has the value HH. REPEAT ('XYZ', 0) has the value of a zero-length string.

13.7.99 RESHAPE (SOURCE, SHAPE [, PAD, ORDER])

Description. Constructs an array of a specified shape from the elements of a given array.

Class. Transformational function.

Arguments.

SOURCE	may be of any type. It shall not be scalar. If PAD is absent or of size zero, the size of SOURCE shall be greater than or equal to PRODUCT (SHAPE). The size of the result is the product of the values of the elements of SHAPE.
SHAPE	shall be of type integer, rank one, and constant size. Its size shall be positive and less than 8. It shall not have an element whose value is negative.
PAD (optional)	shall be of the same type and type parameters as SOURCE. PAD shall not be scalar.
ORDER (optional)	shall be of type integer, shall have the same shape as SHAPE, and its value shall be a permutation of (1, 2, ..., n), where n is the size of SHAPE. If absent, it is as if it were present with value (1, 2, ..., n).

Result Characteristics. The result is an array of shape SHAPE (that is, SHAPE (RESHAPE (SOURCE, SHAPE, PAD, ORDER)) is equal to SHAPE) with the same type and type parameters as SOURCE.

Result Value. The elements of the result, taken in permuted subscript order ORDER (1), ..., ORDER (n), are those of SOURCE in normal array element order followed if necessary by those of PAD in array element order, followed if necessary by additional copies of PAD in array element order.

Examples. RESHAPE ((/ 1, 2, 3, 4, 5, 6 /), (/ 2, 3 /)) has the value $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$. RESHAPE ((/ 1, 2, 3, 4, 5, 6 /), (/ 2, 4 /), (/ 0, 0 /), (/ 2, 1 /)) has the value $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 0 & 0 \end{bmatrix}$.

13.7.100 RRSPACING (X)

Description. Returns the reciprocal of the relative spacing of model numbers near the argument value.

Class. Elemental function.

Argument. X shall be of type real.

Result Characteristics. Same as X.

Result Value. The result has the value $|X \times b^{-e}| \times b^p$, where b, e, and p are as defined in 13.4 for the model representation of X. If X is an IEEE infinity, the result is zero. If X is an IEEE NaN, the result is that NaN.

Example. RRSPACING (–3.0) has the value 0.75×2^{24} for reals whose model is as in Note 13.4.

13.7.101 SAME_TYPE_AS (A, B)

Description. Inquires whether the dynamic type of A is the same as the dynamic type of B.

Class. Inquiry function.

Arguments.

A	shall be an object of extensible type. If it is a pointer, it shall not have an undefined association status.

B — shall be an object of extensible type. If it is a pointer, it shall not have an undefined association status.

Result Characteristics. Default logical scalar.

Result Value. The result is true if and only if the dynamic type of A is the same as the dynamic type of B.

NOTE 13.17

The dynamic type of a disassociated pointer or unallocated allocatable is its declared type.

13.7.102 SCALE (X, I)

Description. Returns $X \times b^{I}$ where b is the base of the model representation of X.

Class. Elemental function.

Arguments.

X	shall be of type real.
I	shall be of type integer.

Result Characteristics. Same as X.

Result Value. The result has the value $X \times b^{I}$, where b is defined in 13.4 for model numbers representing values of X, provided this result is within range; if not, the result is processor dependent.

Example. SCALE (3.0, 2) has the value 12.0 for reals whose model is as in Note 13.4.

13.7.103 SCAN (STRING, SET [, BACK, KIND])

Description. Scan a string for any one of the characters in a set of characters.

Class. Elemental function.

Arguments.

STRING	shall be of type character.
SET	shall be of type character with the same kind type parameter as STRING.
BACK (optional)	shall be of type logical.
KIND (optional)	shall be a scalar integer initialization expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type.

Result Value.

Case (i): If BACK is absent or is present with the value false and if STRING contains at least one character that is in SET, the value of the result is the position of the leftmost character of STRING that is in SET.

Case (ii): If BACK is present with the value true and if STRING contains at least one character that is in SET, the value of the result is the position of the rightmost character of STRING that is in SET.

Case (iii): The value of the result is zero if no character of STRING is in SET or if the length of STRING or SET is zero.

Examples.

Case (i): SCAN ('FORTRAN', 'TR') has the value 3.

Case (ii): SCAN ('FORTRAN', 'TR', BACK = .TRUE.) has the value 5.

Case (iii): SCAN ('FORTRAN', 'BCD') has the value 0.

13.7.104 SELECTED_CHAR_KIND (NAME)

Description. Returns the value of the kind type parameter of the character set named by the argument.

Class. Transformational function.

Argument. NAME shall be scalar and of type default character.

Result Characteristics. Default integer scalar.

Result Value. If NAME has the value DEFAULT, then the result has a value equal to that of the kind type parameter of the default character type. If NAME has the value ASCII, then the result has a value equal to that of the kind type parameter of the ASCII character type if the processor supports such a type; otherwise the result has the value -1. If NAME has the value ISO_10646, then the result has a value equal to that of the kind type parameter of the ISO/IEC 10646-1:2000 UCS-4 character type if the processor supports such a type; otherwise the result has the value -1. If NAME is a processor-defined name of some other character type supported by the processor, then the result has a value equal to that of the kind type parameter of that character type. If NAME is not the name of a supported character type, then the result has the value -1. The NAME is interpreted without respect to case or trailing blanks.

Examples. SELECTED_CHAR_KIND ('ASCII') has the value 1 on a processor that uses 1 as the kind type parameter for the ASCII character set. The following subroutine produces a Japanese date stamp.

```
SUBROUTINE create_date_string(string)
  INTRINSIC date_and_time,selected_char_kind
  INTEGER,PARAMETER :: ucs4 = selected_char_kind("ISO_10646")
  CHARACTER(1,UCS4),PARAMETER :: nen=CHAR(INT(Z'5e74'),UCS4), & !year
    gatsu=CHAR(INT(Z'6708'),UCS4), & !month
    nichi=CHAR(INT(Z'65e5'),UCS4) !day
  CHARACTER(len= *, kind= ucs4) string
  INTEGER values(8)
  CALL date_and_time(values=values)
  WRITE(string,1) values(1),nen,values(2),gatsu,values(3),nichi
1 FORMAT(I0,A,I0,A,I0,A)
END SUBROUTINE"
```

13.7.105 SELECTED_INT_KIND (R)

Description. Returns a value of the kind type parameter of an integer type that represents all integer values n with $-10^{\mathrm{R}} < n < 10^{\mathrm{R}}$.

Class. Transformational function.

Argument. R shall be scalar and of type integer.

Result Characteristics. Default integer scalar.

Result Value. The result has a value equal to the value of the kind type parameter of an integer type that represents all values n in the range $-10^{\mathrm{R}} < n < 10^{\mathrm{R}}$, or if no such kind type parameter is available on the processor, the result is –1. If more than one kind type parameter meets the criterion, the value returned is the one with the smallest decimal exponent range, unless there are several such values, in which case the smallest of these kind values is returned.

Example. Assume a processor supports two integer kinds, 32 with representation method $r = 2$ and $q = 31$, and 64 with representation method $r = 2$ and $q = 63$. On this processor SELECTED_INT_KIND(9) has the value 32 and SELECTED_INT_KIND(10) has the value 64.

13.7.106 SELECTED_REAL_KIND ([P, R])

Description. Returns a value of the kind type parameter of a real type with decimal precision of at least P digits and a decimal exponent range of at least R.

Class. Transformational function.

Arguments. At least one argument shall be present.

P (optional) shall be scalar and of type integer.

R (optional) shall be scalar and of type integer.

Result Characteristics. Default integer scalar.

Result Value. If P or R is absent, the result value is the same as if it were present with the value zero. The result has a value equal to a value of the kind type parameter of a real type with decimal precision, as returned by the function PRECISION, of at least P digits and a decimal exponent range, as returned by the function RANGE, of at least R, or if no such kind type parameter is available on the processor, the result is -1 if the processor does not support a real type with a precision greater than or equal to P but does support a real type with an exponent range greater than or equal to R, -2 if the processor does not support a real type with an exponent range greater than or equal to R but does support a real type with a precision greater than or equal to P, -3 if the processor supports no real type with either of these properties, and -4 if the processor supports real types for each separately but not together. If more than one kind type parameter value meets the criteria, the value returned is the one with the smallest decimal precision, unless there are several such values, in which case the smallest of these kind values is returned.

Example. SELECTED_REAL_KIND (6, 70) has the value KIND (0.0) on a machine that supports a default real approximation method with $b = 16$, $p = 6$, $e_{\min} = -64$, and $e_{\max} = 63$ and does not have a less precise approximation method.

13.7.107 SET_EXPONENT (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.

Class. Elemental function.

Arguments.

X shall be of type real.

I shall be of type integer.

Result Characteristics. Same as X.

Result Value. The result has the value X $\times b^{\mathrm{I}-e}$, where b and e are as defined in 13.4 for the model representation of X. If X has value zero, the result has value zero.

Example. SET_EXPONENT (3.0, 1) has the value 1.5 for reals whose model is as in Note 13.4.

13.7.108 SHAPE (SOURCE [, KIND])

Description. Returns the shape of an array or a scalar.

Class. Inquiry function.

Arguments.

SOURCE	may be of any type. It may be a scalar or an array. It shall not be an unallocated allocatable or a pointer that is not associated. It shall not be an assumed-size array.
KIND (optional)	shall be a scalar integer initialization expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type. The result is an array of rank one whose size is equal to the rank of SOURCE.

Result Value. The value of the result is the shape of SOURCE.

Examples. The value of SHAPE (A (2:5, –1:1)) is [4, 3]. The value of SHAPE (3) is the rank-one array of size zero.

13.7.109 SIGN (A, B)

Description. Magnitude of A with the sign of B.

Class. Elemental function.

Arguments.

A	shall be of type integer or real.
B	shall be of the same type and kind type parameter as A.

Result Characteristics. Same as A.

Result Value.

Case (i): If B $>$ 0, the value of the result is $|$A$|$.

Case (ii): If B $<$ 0, the value of the result is -$|$A$|$.

Case (iii): If B is of type integer and B=0, the value of the result is $|$A$|$.

Case (iv): If B is of type real and is zero, then

(1) If the processor cannot distinguish between positive and negative real zero, the value of the result is $|$A$|$.

(2) If B is positive real zero, the value of the result is $|$A$|$.

(3) If B is negative real zero, the value of the result is -|A|.

Example. SIGN (–3.0, 2.0) has the value 3.0.

13.7.110 SIN (X)

Description. Sine function.

Class. Elemental function.

Argument. X shall be of type real or complex.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to sin(X). If X is of type real, 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).

13.7.111 SINH (X)

Description. Hyperbolic sine function.

Class. Elemental function.

Argument. X shall be of type real.

Result Characteristics. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to sinh(X).

Example. SINH (1.0) has the value 1.1752012 (approximately).

13.7.112 SIZE (ARRAY [, DIM, KIND])

Description. Returns the extent of an array along a specified dimension or the total number of elements in the array.

Class. Inquiry function.

Arguments.

ARRAY	may be of any type. It shall not be scalar. It shall not be an unallocated allocatable or a pointer that is not associated. If ARRAY is an assumed-size array, DIM shall be present with a value less than the rank of ARRAY.
DIM (optional)	shall be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY.
KIND (optional)	shall be a scalar integer initialization expression.

Result Characteristics. Integer scalar. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type.

Result Value. The result has a value equal to the extent of dimension DIM of ARRAY or, if DIM is absent, the total number of elements of ARRAY.

Examples. The value of SIZE (A (2:5, −1:1), DIM=2) is 3. The value of SIZE (A (2:5, −1:1)) is 12.

13.7.113 SPACING (X)

Description. Returns the absolute spacing of model numbers near the argument value.

Class. Elemental function.

Argument. X shall be of type real.

Result Characteristics. Same as X.

Result Value. If X does not have the value zero, the result has the value $b^{\max(e-p, e_{\text{MIN}}-1)}$, where b, e, and p are as defined in 13.4 for the model representation of X. If X has the value zero, the result is the same as that of TINY (X). If X is an IEEE infinity, the result is positive infinity. If X is an IEEE NaN, the result is that NaN.

Example. SPACING (3.0) has the value 2^{-22} for reals whose model is as in Note 13.4.

13.7.114 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.

Class. Transformational function.

Arguments.

SOURCE	may be of any type. It may be a scalar or an array. The rank of SOURCE shall be less than 7.
DIM	shall 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	shall be scalar and of type integer.

Result Characteristics. 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 an array with shape (d_1, d_2, ..., d_n), the shape of the result is (d_1, d_2, ..., $d_{\text{DIM}-1}$, MAX (NCOPIES, 0), d_{DIM}, ..., d_n).

Result Value.

Case (i): If SOURCE is scalar, each element of the result has a value equal to SOURCE.

Case (ii): If SOURCE is an array, the element of the result with subscripts (r_1, r_2, ..., r_{n+1}) has the value SOURCE (r_1, r_2, ..., $r_{\text{DIM}-1}$, $r_{\text{DIM}+1}$, ..., r_{n+1}).

Examples. If A is the array [2, 3, 4], SPREAD (A, DIM=1, NCOPIES=NC) is the array $\begin{bmatrix} 2 & 3 & 4 \\ 2 & 3 & 4 \\ 2 & 3 & 4 \end{bmatrix}$ if NC has the value 3 and is a zero-sized array if NC has the value 0.

13.7.115 SQRT (X)

Description. Square root.

Class. Elemental function.

Argument. X shall be of type real or complex. Unless X is complex, its value shall be greater than or equal to zero.

Result Characteristics. Same as X.

Result Value. The result has a 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 has the same sign as the imaginary part of X.

Example. SQRT (4.0) has the value 2.0 (approximately).

13.7.116 SUM (ARRAY, DIM [, MASK]) or SUM (ARRAY [, MASK])

Description. Sum all the elements of ARRAY along dimension DIM corresponding to the true elements of MASK.

Class. Transformational function.

Arguments.

ARRAY	shall be of type integer, real, or complex. It shall not be scalar.
DIM	shall be scalar and of type integer with a value in the range $1 \le \text{DIM} \le n$, where n is the rank of ARRAY. The corresponding actual argument shall not be an optional dummy argument.
MASK (optional)	shall be of type logical and shall be conformable with ARRAY.

Result Characteristics. The result is of the same type and kind type parameter as ARRAY. It is scalar if DIM is absent; otherwise, the result has rank $n - 1$ and 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 a value equal to a processor-dependent approximation to the sum of all the elements of ARRAY or has the value zero if ARRAY has size zero.

Case (ii): The result of SUM (ARRAY, MASK = MASK) has a value equal to a processor-dependent approximation to the sum of the elements of ARRAY corresponding to the true elements of MASK or has the value zero if there are no true elements.

Case (iii): If ARRAY has rank one, SUM (ARRAY, DIM = DIM [, MASK = MASK]) has a value equal to that of SUM (ARRAY [,MASK = MASK]). Otherwise, the value of element (s_1, s_2, ..., $s_{\text{DIM}-1}$, $s_{\text{DIM}+1}$, ..., s_n) SUM (ARRAY, DIM = DIM [, MASK = MASK]) is equal to

SUM (ARRAY (s_1, s_2, ..., $s_{\text{DIM}-1}$, :, $s_{\text{DIM}+1}$, ..., s_n) [, MASK= 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 > 0.0) forms the 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 $[9,\ 12]$.

13.7.117 SYSTEM_CLOCK ([COUNT, COUNT_RATE, COUNT_MAX])

Description. Returns numeric data from a real-time clock.

Class. Subroutine.

Arguments.

COUNT (optional)	shall be scalar and of type integer. It is an INTENT (OUT) argument. It is assigned a processor-dependent value based on the current value of the processor clock, or –HUGE (COUNT) if there is no clock. The processor-

Example. TANH (1.0) has the value 0.76159416 (approximately).

13.7.120 TINY (X)

Description. Returns the smallest positive number of the model representing numbers of the same type and kind type parameter as the argument.

Class. Inquiry function.

Argument. X shall be of type real. It may be a scalar or an array.

Result Characteristics. Scalar with the same type and kind type parameter as X.

Result Value. The result has the value $b^{e_{\min}-1}$ where b and $e_{\min}$ are as defined in 13.4 for the model representing numbers of the same type and kind type parameter as X.

Example. TINY (X) has the value 2^{-127} for real X whose model is as in Note 13.4.

13.7.121 TRANSFER (SOURCE, MOLD [, SIZE])

Description. Returns a result with a physical representation identical to that of SOURCE but interpreted with the type and type parameters of MOLD.

Class. Transformational function.

Arguments.

SOURCE	may be of any type. It may be a scalar or an array.
MOLD	may be of any type. It may be a scalar or an array. If it is a variable, it need not be defined.
SIZE (optional)	shall be scalar and of type integer. The corresponding actual argument shall not be an optional dummy argument.

Result Characteristics. The result is of the same type and type parameters as MOLD.

Case (i): If MOLD is a scalar and SIZE is absent, the result is a scalar.

Case (ii): If MOLD is an array and SIZE is absent, the result is an array and of rank one. Its size is as small as possible such that its physical representation is not shorter than that of SOURCE.

Case (iii): If SIZE is present, the result is an array of rank one and size SIZE.

Result Value. If the physical representation of the result has the same length as that of SOURCE, the physical representation of the result is that of SOURCE. If the physical representation of the result is longer than that of SOURCE, the physical representation of the leading part is that of SOURCE and the remainder is processor dependent. If the physical representation of the result is shorter than that of SOURCE, the physical representation of the result is the leading part of SOURCE. If D and E are scalar variables such that the physical representation of D is as long as or longer than that of E, the value of TRANSFER (TRANSFER (E, D), E) shall be the value of E. IF D is an array and E is an array of rank one, the value of TRANSFER (TRANSFER (E, D), E, SIZE (E)) shall be the value of E.

Examples.

Case (i): TRANSFER (1082130432, 0.0) has the value 4.0 on a processor that represents the values 4.0 and 1082130432 as the string of binary digits 0100 0000 1000 0000 0000 0000 0000 0000.

Case (ii): TRANSFER ((/ 1.1, 2.2, 3.3 /), (/ (0.0, 0.0) /)) is a complex rank-one array of length two whose first element has the value (1.1, 2.2) and whose second element has a real part with the value 3.3. The imaginary part of the second element is processor dependent.

Case (iii): TRANSFER ((/ 1.1, 2.2, 3.3 /), (/ (0.0, 0.0) /), 1) is a complex rank-one array of length one whose only element has the value (1.1, 2.2).

13.7.122 TRANSPOSE (MATRIX)

Description. Transpose an array of rank two.

Class. Transformational function.

Argument. MATRIX may be of any type and shall have rank two.

Result Characteristics. 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 the 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 $\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$.

13.7.123 TRIM (STRING)

Description. Returns the argument with trailing blank characters removed.

Class. Transformational function.

Argument. STRING shall be of type character and shall be a scalar.

Result Characteristics. Character with the same kind type parameter value as STRING and 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 the value ' A B'.

13.7.124 UBOUND (ARRAY [, DIM, KIND])

Description. Returns all the upper bounds of an array or a specified upper bound.

Class. Inquiry function.

Arguments.

ARRAY may be of any type. It shall not be scalar. It shall not be an unallocated allocatable or a pointer that is not associated. If ARRAY is an assumed-size array, DIM shall be present with a value less than the rank of ARRAY.

DIM (optional) shall be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY. The corresponding actual argument shall not be an optional dummy argument.

KIND (optional) shall be a scalar integer initialization expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type. 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): For an array section or for an array expression, other than a whole array or array structure component, UBOUND (ARRAY, DIM) has a value equal to the number of elements in the given dimension; otherwise, it has a value equal to the 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.

Case (ii): UBOUND (ARRAY) has a value whose ith element is equal to UBOUND (ARRAY, i), for $i = 1, 2, ..., n$, where n is the rank of ARRAY.

Examples. If A is declared by the statement

```
REAL A (2:3, 7:10)
```

then UBOUND (A) is [3, 10] and UBOUND (A, DIM = 2) is 10.

13.7.125 UNPACK (VECTOR, MASK, FIELD)

Description. Unpack an array of rank one into an array under the control of a mask.

Class. Transformational function.

Arguments.

VECTOR	may be of any type. It shall have rank one. Its size shall be at least t where t is the number of true elements in MASK.
MASK	shall be an array of type logical.
FIELD	shall be of the same type and type parameters as VECTOR and shall be conformable with MASK.

Result Characteristics. 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 ith true element of MASK, in array element order, has the value VECTOR (i) for $i = 1, 2, ..., t$, where t is the number of true values in MASK. Each other element has a value equal to FIELD if FIELD is scalar or to the corresponding element of FIELD if it is an array.

Examples. Particular values may be "scattered" to particular positions in an array by using UNPACK. If M is the array $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, 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 $\begin{bmatrix} 1 & 2 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 3 \end{bmatrix}$ and the result of UN-

PACK (V, MASK = Q, FIELD = 0) has the value $\begin{bmatrix} 0 & 2 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 3 \end{bmatrix}$.

13.7.126 VERIFY (STRING, SET [, BACK, KIND])

Description. Verify that a set of characters contains all the characters in a string by identifying the position of the first character in a string of characters that does not appear in a given set of characters.

Class. Elemental function.

Arguments.

STRING	shall be of type character.
SET	shall be of type character with the same kind type parameter as STRING.
BACK (optional)	shall be of type logical.
KIND (optional)	shall be a scalar integer initialization expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type.

Result Value.

Case (i): If BACK is absent or has the value false and if STRING contains at least one character that is not in SET, the value of the result is the position of the leftmost character of STRING that is not in SET.

Case (ii): If BACK is present with the value true and if STRING contains at least one character that is not in SET, the value of the result is the position of the rightmost character of STRING that is not in SET.

Case (iii): The value of the result is zero if each character in STRING is in SET or if STRING has zero length.

Examples.

Case (i): VERIFY ('ABBA', 'A') has the value 2.

Case (ii): VERIFY ('ABBA', 'A', BACK = .TRUE.) has the value 3.

Case (iii): VERIFY ('ABBA', 'AB') has the value 0.

13.8 Standard intrinsic modules

This standard defines several intrinsic modules. A processor may extend the standard intrinsic modules to provide public entities in them in addition to those specified in this standard.

NOTE 13.18

To avoid potential name conflicts with program entities, it is recommended that a program use the ONLY option in any USE statement that accesses a standard intrinsic module.

13.8.1 The IEEE modules

The IEEE_EXCEPTIONS, IEEE_ARITHMETIC, and IEEE_FEATURES intrinsic modules are described in Section 14.

13.8.2 The ISO_FORTRAN_ENV intrinsic module

The intrinsic module ISO_FORTRAN_ENV provides public entities relating to the Fortran environment.

The processor shall provide the named constants described in the following subclauses.

13.8.2.1 CHARACTER_STORAGE_SIZE

The value of the default integer scalar constant CHARACTER_STORAGE_SIZE is the size expressed in bits of the character storage unit (16.4.3.1).

13.8.2.2 ERROR_UNIT

The value of the default integer scalar constant ERROR_UNIT identifies the processor-dependent preconnected external unit used for the purpose of error reporting (9.4). This unit may be the same as OUTPUT_UNIT. The value shall not be -1.

13.8.2.3 FILE_STORAGE_SIZE

The value of the default integer scalar constant FILE_STORAGE_SIZE is the size expressed in bits of the file storage unit (9.2.4).

13.8.2.4 INPUT_UNIT

The value of the default integer scalar constant INPUT_UNIT identifies the same processor-dependent preconnected external unit as the one identified by an asterisk in a READ statement (9.4). The value shall not be -1.

13.8.2.5 IOSTAT_END

The value of the default integer scalar constant IOSTAT_END is assigned to the variable specified in an IOSTAT= specifier (9.10.4) if an end-of-file condition occurs during execution of an input/output statement and no error condition occurs. This value shall be negative.

13.8.2.6 IOSTAT_EOR

The value of the default integer scalar constant IOSTAT_EOR is assigned to the variable specified in an IOSTAT= specifier (9.10.4) if an end-of-record condition occurs during execution of an input/output statement and no end-of-file or error condition occurs. This value shall be negative and different from the value of IOSTAT_END.

13.8.2.7 NUMERIC_STORAGE_SIZE

The value of the default integer scalar constant NUMERIC_STORAGE_SIZE is the size expressed in bits of the numeric storage unit (16.4.3.1).

13.8.2.8 OUTPUT_UNIT

The value of the default integer scalar constant OUTPUT_UNIT identifies the same processor-dependent preconnected external unit as the one identified by an asterisk in a WRITE statement (9.4). The value shall not be -1.

13.8.3 The ISO_C_BINDING module

The ISO_C_BINDING intrinsic module is described in Section 15.

Section 14: Exceptions and IEEE arithmetic

The intrinsic modules IEEE_EXCEPTIONS, IEEE_ARITHMETIC, and IEEE_FEATURES provide support for exceptions and IEEE arithmetic. Whether the modules are provided is processor dependent. If the module IEEE_FEATURES is provided, which of the named constants defined in this standard are included is processor dependent. The module IEEE_ARITHMETIC behaves as if it contained a USE statement for IEEE_EXCEPTIONS; everything that is public in IEEE_EXCEPTIONS is public in IEEE_ARITHMETIC.

NOTE 14.1

> The types and procedures defined in these modules are not themselves intrinsic.

If IEEE_EXCEPTIONS or IEEE_ARITHMETIC is accessible in a scoping unit, IEEE_OVERFLOW and IEEE_DIVIDE_BY_ZERO are supported in the scoping unit for all kinds of real and complex data. Which other exceptions are supported can be determined by the function IEEE_SUPPORT_FLAG (14.10.26); whether control of halting is supported can be determined by the function IEEE_SUPPORT_HALTING. The extent of support of the other exceptions may be influenced by the accessibility of the named constants IEEE_INEXACT_FLAG, IEEE_INVALID_FLAG, and IEEE_UNDERFLOW_FLAG of the module IEEE_FEATURES. If a scoping unit has access to IEEE_UNDERFLOW_FLAG of IEEE_FEATURES, within the scoping unit the processor shall support underflow and return true from IEEE_SUPPORT_FLAG(IEEE_UNDERFLOW, X) for at least one kind of real. Similarly, if IEEE_INEXACT_FLAG or IEEE_INVALID_FLAG is accessible, within the scoping unit the processor shall support the exception and return true from the corresponding inquiry for at least one kind of real. Also, if IEEE_HALTING is accessible, within the scoping unit the processor shall support control of halting and return true from IEEE_SUPPORT_HALTING(FLAG) for the flag.

NOTE 14.2

> IEEE_INVALID is not required to be supported whenever IEEE_EXCEPTIONS is accessed. This is to allow a non-IEEE processor to provide support for overflow and divide_by_zero. On an IEEE machine, invalid is an equally serious condition.

NOTE 14.3

> The IEEE_FEATURES module is provided to allow a reasonable amount of cooperation between the programmer and the processor in controlling the extent of IEEE arithmetic support. On some processors some IEEE features are natural for the processor to support, others may be inefficient at run time, and others are essentially impossible to support. If IEEE_FEATURES is not used, the processor will support only the natural operations. Within IEEE_FEATURES the processor will define the named constants (14.1) corresponding to the time-consuming features (as well as the natural ones for completeness) but will not define named constants corresponding to the impossible features. If the programmer accesses IEEE_FEATURES, the processor shall provide support for all of the IEEE_FEATURES that are reasonably possible. If the programmer uses an ONLY clause on a USE statement to access a particular feature name, the processor shall provide support for the corresponding feature, or issue an error message saying the name is not defined in the module.
>
> When used this way, the named constants in the IEEE_FEATURES are similar to what are frequently called command line switches for the compiler. They can specify compilation options in a reasonably portable manner.

If a scoping unit does not access IEEE_FEATURES, IEEE_EXCEPTIONS, or IEEE_ARITHMETIC, the level of support is processor dependent, and need not include support for any exceptions. If a flag is signaling on entry to such a scoping unit, the processor ensures that it is signaling on exit. If a flag is quiet on entry to such a scoping unit, whether it is signaling on exit is processor dependent.

Further IEEE support is available through the module IEEE_ARITHMETIC. The extent of support may be influenced by the accessibility of the named constants of the module IEEE_FEATURES. If a scoping unit has access to IEEE_DATATYPE of IEEE_FEATURES, within the scoping unit the processor shall support IEEE arithmetic and return true from IEEE_SUPPORT_DATATYPE(X) (14.10.23) for at least one kind of real. Similarly, if IEEE_DENORMAL, IEEE_DIVIDE, IEEE_INF, IEEE_NAN, IEEE_ROUNDING, or IEEE_SQRT is accessible, within the scoping unit the processor shall support the feature and return true from the corresponding inquiry function for at least one kind of real. In the case of IEEE_ROUNDING, it shall return true for all the rounding modes IEEE_NEAREST, IEEE_TO_ZERO, IEEE_UP, and IEEE_DOWN.

Execution might be slowed on some processors by the support of some features. If IEEE_EXCEPTIONS or IEEE_ARITHMETIC is accessed but IEEE_FEATURES is not accessed, the supported subset of features is processor dependent. The processor's fullest support is provided when all of IEEE_FEATURES is accessed as in

```
USE, INTRINSIC :: IEEE_ARITHMETIC; USE, INTRINSIC :: IEEE_FEATURES
```

but execution might then be slowed by the presence of a feature that is not needed. In all cases, the extent of support can be determined by the inquiry functions.

14.1 Derived types and constants defined in the modules

The modules IEEE_EXCEPTIONS, IEEE_ARITHMETIC, and IEEE_FEATURES define five derived types, whose components are all private.

The module IEEE_EXCEPTIONS defines

- IEEE_FLAG_TYPE, for identifying a particular exception flag. Its only possible values are those of named constants defined in the module: IEEE_INVALID, IEEE_OVERFLOW, IEEE_DIVIDE_-BY_ZERO, IEEE_UNDERFLOW, and IEEE_INEXACT. The module also defines the array named constants IEEE_USUAL = (/ IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_INVALID /) and IEEE_ALL = (/ IEEE_USUAL, IEEE_UNDERFLOW, IEEE_INEXACT /).
- IEEE_STATUS_TYPE, for saving the current floating point status.

The module IEEE_ARITHMETIC defines

- IEEE_CLASS_TYPE, for identifying a class of floating-point values. Its only possible values are those of named constants defined in the module: IEEE_SIGNALING_NAN, IEEE_QUIET_NAN, IEEE_NEGATIVE_INF, IEEE_NEGATIVE_NORMAL, IEEE_NEGATIVE_DENORMAL, IEEE_NEGATIVE_ZERO, IEEE_POSITIVE_ZERO, IEEE_POSITIVE_DENORMAL, IEEE_-POSITIVE_NORMAL, IEEE_POSITIVE_INF, and IEEE_OTHER_VALUE.
- IEEE_ROUND_TYPE, for identifying a particular rounding mode. Its only possible values are those of named constants defined in the module: IEEE_NEAREST, IEEE_TO_ZERO, IEEE_UP, and IEEE_DOWN for the IEEE modes; and IEEE_OTHER for any other mode.

- The elemental operator == for two values of one of these types to return true if the values are the same and false otherwise.
- The elemental operator /= for two values of one of these types to return true if the values differ and false otherwise.

The module IEEE_FEATURES defines

- IEEE_FEATURES_TYPE, for expressing the need for particular IEEE features. Its only possible values are those of named constants defined in the module: IEEE_DATATYPE, IEEE_DENORMAL, IEEE_DIVIDE, IEEE_HALTING, IEEE_INEXACT_FLAG, IEEE_INF, IEEE_INVALID_FLAG, IEEE_NAN, IEEE_ROUNDING, IEEE_SQRT, and IEEE_UNDERFLOW_FLAG.

14.2 The exceptions

The exceptions are

- IEEE_OVERFLOW

  This exception occurs when the result for an intrinsic real operation or assignment has an absolute value greater than a processor-dependent limit, or the real or imaginary part of the result for an intrinsic complex operation or assignment has an absolute value greater than a processor-dependent limit.

- IEEE_DIVIDE_BY_ZERO

  This exception occurs when a real or complex division has a nonzero numerator and a zero denominator.

- IEEE_INVALID

  This exception occurs when a real or complex operation or assignment is invalid; possible examples are SQRT(X) when X is real and has a nonzero negative value, and conversion to an integer (by assignment, an intrinsic procedure, or a procedure defined in an intrinsic module) when the result is too large to be representable.

- IEEE_UNDERFLOW

  This exception occurs when the result for an intrinsic real operation or assignment has an absolute value less than a processor-dependent limit and loss of accuracy is detected, or the real or imaginary part of the result for an intrinsic complex operation or assignment has an absolute value less than a processor-dependent limit and loss of accuracy is detected.

- IEEE_INEXACT

  This exception occurs when the result of a real or complex operation or assignment is not exact.

Each exception has a flag whose value is either quiet or signaling. The value can be determined by the function IEEE_GET_FLAG. Its initial value is quiet and it signals when the associated exception occurs. Its status can also be changed by the subroutine IEEE_SET_FLAG or the subroutine IEEE_SET_STATUS. Once signaling within a procedure, it remains signaling unless set quiet by an invocation of the subroutine IEEE_SET_FLAG or the subroutine IEEE_SET_STATUS.

If a flag is signaling on entry to a procedure, the processor will set it to quiet on entry and restore it to signaling on return.

NOTE 14.4

If a flag signals during execution of a procedure, the processor shall not set it to quiet on return.

Evaluation of a specification expression may cause an exception to signal.

In a scoping unit that has access to IEEE_EXCEPTIONS or IEEE_ARITHMETIC, if an intrinsic procedure or a procedure defined in an intrinsic module executes normally, the values of the flags IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, and IEEE_INVALID shall be as on entry to the procedure, even if one or more signals during the calculation. If a real or complex result is too large for the procedure to handle, IEEE_OVERFLOW may signal. If a real or complex result is a NaN because of an invalid operation (for example, LOG(-1.0)), IEEE_INVALID may signal. Similar rules apply to format processing and to intrinsic operations: no signaling flag shall be set quiet and no quiet flag shall be set signaling because of an intermediate calculation that does not affect the result.

NOTE 14.5

An implementation may provide alternative versions of an intrinsic procedure; a practical example of such alternatives might be one version suitable for a call from a scoping unit with access to IEEE_EXCEPTIONS or IEEE_ARITHMETIC and one for other cases.

In a sequence of statements that has no invocations of IEEE_GET_FLAG, IEEE_SET_FLAG, IEEE_GET_STATUS, IEEE_SET_HALTING, or IEEE_SET_STATUS, if the execution of an operation would cause an exception to signal but after execution of the sequence no value of a variable depends on the operation, whether the exception is signaling is processor dependent. For example, when Y has the value zero, whether the code

```
X = 1.0/Y
X = 3.0
```

signals IEEE_DIVIDE_BY_ZERO is processor dependent. Another example is the following:

```
REAL, PARAMETER :: X=0.0, Y=6.0
IF (1.0/X == Y) PRINT *,'Hello world'
```

where the processor is permitted to discard the IF statement because the logical expression can never be true and no value of a variable depends on it.

An exception shall not signal if this could arise only during execution of an operation beyond those required or permitted by the standard. For example, the statement

```
IF (F(X)>0.0) Y = 1.0/Z
```

shall not signal IEEE_DIVIDE_BY_ZERO when both F(X) and Z are zero and the statement

```
WHERE(A>0.0) A = 1.0/A
```

shall not signal IEEE_DIVIDE_BY_ZERO. On the other hand, when X has the value 1.0 and Y has the value 0.0, the expression

```
X>0.00001 .OR. X/Y>0.00001
```

is permitted to cause the signaling of IEEE_DIVIDE_BY_ZERO.

The processor need not support IEEE_INVALID, IEEE_UNDERFLOW, and IEEE_INEXACT. If an exception is not supported, its flag is always quiet. The function IEEE_SUPPORT_FLAG can be used to inquire whether a particular flag is supported.

14.3 The rounding modes

The IEEE standard specifies four rounding modes:

- IEEE_NEAREST rounds the exact result to the nearest representable value.
- IEEE_TO_ZERO rounds the exact result towards zero to the next representable value.
- IEEE_UP rounds the exact result towards +infinity to the next representable value.
- IEEE_DOWN rounds the exact result towards -infinity to the next representable value.

The function IEEE_GET_ROUNDING_MODE can be used to inquire which rounding mode is in operation. Its value is one of the above four or IEEE_OTHER if the rounding mode does not conform to the IEEE standard.

If the processor supports the alteration of the rounding mode during execution, the subroutine IEEE_SET_ROUNDING_MODE can be used to alter it. The function IEEE_SUPPORT_ROUNDING can be used to inquire whether this facility is available for a particular mode. The function IEEE_SUPPORT_IO can be used to inquire whether rounding for base conversion in formatted input/output (9.4.5.13, 9.5.1.12, 10.6.1.2.6) is as specified in the IEEE standard.

In a procedure other than IEEE_SET_ROUNDING_MODE or IEEE_SET_STATUS, the processor shall not change the rounding mode on entry, and on return shall ensure that the rounding mode is the same as it was on entry.

NOTE 14.6

Within a program, all literal constants that have the same form have the same value (4.1.2). Therefore, the value of a literal constant is not affected by the rounding mode.

14.4 Underflow mode

Some processors allow control during program execution of whether underflow produces a denormalized number in conformance with the IEEE standard (gradual underflow) or produces zero instead (abrupt underflow). On some processors, floating-point performance is typically better in abrupt underflow mode than in gradual underflow mode.

Control over the underflow mode is exercised by invocation of IEEE_SET_UNDERFLOW_MODE. The function IEEE_GET_UNDERFLOW_MODE can be used to inquire which underflow mode is in operation. The function IEEE_SUPPORT_UNDERFLOW_MODE can be used to inquire whether this facility is available. The initial underflow mode is processor dependent. In a procedure other than IEEE_SET_UNDERFLOW_MODE or IEEE_SET_STATUS, the processor shall not change the underflow mode on entry, and on return shall ensure that the underflow mode is the same as it was on entry.

The underflow mode affects only floating-point calculations whose type is that of an X for which IEEE_SUPPORT_UNDERFLOW_CONTROL returns true."

14.5 Halting

Some processors allow control during program execution of whether to abort or continue execution after an exception. Such control is exercised by invocation of the subroutine IEEE_SET_HALTING_MODE. Halting is not precise and may occur any time after the exception has occurred. The function IEEE_SUPPORT_HALTING can be used to inquire whether this facility is available. The initial halting mode is processor dependent. In a procedure other than IEEE_SET_HALTING_MODE or IEEE_SET_STATUS, the processor shall not change the halting mode on entry, and on return shall ensure that the halting mode is the same as it was on entry.

14.6 The floating point status

The values of all the supported flags for exceptions, rounding mode, underflow mode, and halting are called the floating point status. The floating point status can be saved in a scalar variable of type TYPE(IEEE_STATUS_TYPE) with the subroutine IEEE_GET_STATUS and restored with the subroutine IEEE_SET_ but the normalized numbers shall be exactly those of an IEEE floating-point format; the operations of addition, subtraction, and multiplication shall be implemented with at least one of the IEEE rounding modes; the IEEE operation rem shall be provided by the function IEEE_REM; and the IEEE functions copysign, scalb, logb, nextafter, and unordered shall be provided by the functions IEEE_COPY_SIGN, IEEE_SCALB, IEEE_LOGB, IEEE_NEXT_AFTER, and IEEE_UNORDERED, respectively. The inquiry function IEEE_SUPPORT_DIVIDE is provided to inquire whether the processor supports divide with the accuracy specified by the IEEE standard. For each of the operations of addition, subtraction, and multiplication, the result shall be as specified in the IEEE standard whenever the IEEE result is normalized and the operands are normalized (if floating point) or are valid and within range (if another type).

The inquiry function IEEE_SUPPORT_NAN is provided to inquire whether the processor supports IEEE NaNs. Where these are supported, their behavior for unary and binary operations, including those defined by intrinsic functions and by functions in intrinsic modules, shall be consistent with the specifications in the IEEE standard.

The inquiry function IEEE_SUPPORT_INF is provided to inquire whether the processor supports IEEE infinities. Where these are supported, their behavior for unary and binary operations, including those defined by intrinsic functions and by functions in intrinsic modules, shall be consistent with the specifications in the IEEE standard.

The inquiry function IEEE_SUPPORT_INF is provided to inquire whether the processor supports IEEE infinities. Where these are supported, their behavior for unary and binary operations, including those defined by intrinsic functions and by functions in intrinsic modules, shall be consistent with the specifications in the IEEE standard.

The inquiry function IEEE_SUPPORT_DENORMAL is provided to inquire whether the processor supports IEEE denormals. Where these are supported, their behavior for unary and binary operations, including those defined by intrinsic functions and by functions in intrinsic modules, shall be consistent with the specifications in the IEEE standard.

The IEEE standard specifies a square root function that returns -0.0 for the square root of -0.0 and has certain accuracy requirements. The function IEEE_SUPPORT_SQRT can be used to inquire whether SQRT is implemented in accord with the IEEE standard for a particular kind of real.

The inquiry function IEEE_SUPPORT_STANDARD is provided to inquire whether the processor supports all the IEEE facilities defined in this standard for a particular kind of real.

14.9 Tables of the procedures

For all of the procedures defined in the modules, the arguments shown are the names that shall be used for argument keywords if the keyword form is used for the actual arguments.

The procedure classification terms "inquiry function" and "transformational function" are used here with the same meanings as in 13.1.

14.9.1 Inquiry functions

The module IEEE_EXCEPTIONS contains the following inquiry functions:

IEEE_SUPPORT_FLAG (FLAG [, X])	Are IEEE exceptions supported?
IEEE_SUPPORT_HALTING (FLAG)	Is IEEE halting control supported?

The module IEEE_ARITHMETIC contains the following inquiry functions:

IEEE_SUPPORT_DATATYPE ([X])	Is IEEE arithmetic supported?
IEEE_SUPPORT_DENORMAL ([X])	Are IEEE denormalized numbers supported?
IEEE_SUPPORT_DIVIDE ([X])	Is IEEE divide supported?
IEEE_SUPPORT_INF ([X])	Is IEEE infinity supported?
IEEE_SUPPORT_IO ([X])	Is IEEE formatting supported?
IEEE_SUPPORT_NAN ([X])	Are IEEE NaNs supported?
IEEE_SUPPORT_ROUNDING (ROUND_VALUE [, X])	Is IEEE rounding supported?
IEEE_SUPPORT_SQRT ([X])	Is IEEE square root supported?
IEEE_SUPPORT_STANDARD ([X])	Are all IEEE facilities supported?
IEEE_SUPPORT_UNDERFLOW_CONTROL ([X])	Is IEEE underflow control supported?

14.9.2 Elemental functions

The module IEEE_ARITHMETIC contains the following elemental functions for reals X and Y for which IEEE_SUPPORT_DATATYPE(X) and IEEE_SUPPORT_DATATYPE(Y) are true:

IEEE_CLASS (X)	IEEE class.
IEEE_COPY_SIGN (X,Y)	IEEE copysign function.
IEEE_IS_FINITE (X)	Determine if value is finite.
IEEE_IS_NAN (X)	Determine if value is IEEE Not-a-Number.
IEEE_IS_NORMAL (X)	Determine if a value is normal, that is, neither an infinity, a NaN, nor denormalized.
IEEE_IS_NEGATIVE (X)	Determine if value is negative.
IEEE_LOGB (X)	Unbiased exponent in the IEEE floating point format.
IEEE_NEXT_AFTER (X,Y)	Returns the next representable neighbor of X in the direction toward Y.
IEEE_REM (X,Y)	The IEEE REM function, that is X - Y*N, where N is the integer nearest to the exact value X/Y.
IEEE_RINT (X)	Round to an integer value according to the current rounding mode.
IEEE_SCALB (X,I)	Returns $X \times 2^I$.
IEEE_UNORDERED (X,Y)	IEEE unordered function. True if X or Y is a NaN and false otherwise.
IEEE_VALUE (X,CLASS)	Generate an IEEE value.

14.9.3 Kind function

The module IEEE_ARITHMETIC contains the following transformational function:

IEEE_SELECTED_REAL_KIND ([P, R])	Kind type parameter value for an IEEE real with given precision and range.

14.9.4 Elemental subroutines

The module IEEE_EXCEPTIONS contains the following elemental subroutines:

IEEE_GET_FLAG (FLAG,FLAG_VALUE)	Get an exception flag.
IEEE_GET_HALTING_MODE (FLAG, HALTING)	Get halting mode for an exception.

14.9.5 Nonelemental subroutines

The module IEEE_EXCEPTIONS contains the following nonelemental subroutines:

IEEE_GET_STATUS (STATUS_VALUE)	Get the current state of the floating point environment.
IEEE_SET_FLAG (FLAG,FLAG_VALUE)	Set an exception flag.
IEEE_SET_HALTING_MODE (FLAG, HALTING)	Controls continuation or halting on exceptions.
IEEE_SET_STATUS (STATUS_VALUE)	Restore the state of the floating point environment.

The module IEEE_ARITHMETIC contains the following nonelemental subroutines:

IEEE_GET_ROUNDING_MODE (ROUND_VALUE)	Get the current IEEE rounding mode.
IEEE_GET_UNDERFLOW_MODE (GRADUAL)	Get the current underflow mode.
IEEE_SET_ROUNDING_MODE (ROUND_VALUE)	Set the current IEEE rounding mode.
IEEE_SET_UNDERFLOW_MODE (GRADUAL)	Set the current underflow mode.

14.10 Specifications of the procedures

In the detailed descriptions below, procedure names are generic and are not specific. All the functions are pure. The dummy arguments of the intrinsic module procedures in 14.9.1, 14.9.2, and 14.9.3 have INTENT(IN). The dummy arguments of the intrinsic module procedures in 14.9.4 and 14.9.5 have INTENT(IN) if the intent is not stated explicitly. In the examples, it is assumed that the processor supports IEEE arithmetic for default real.

NOTE 14.9

It is intended that a processor should not check a condition given in a paragraph labeled "**Restriction**" at compile time, but rather should rely on the programmer writing code such as

```
IF (IEEE_SUPPORT_DATATYPE(X)) THEN
  C = IEEE_CLASS(X)
ELSE
  .
  .
ENDIF
```

to avoid a call being made on a processor for which the condition is violated.

For the elemental functions of IEEE_ARITHMETIC, as tabulated in 14.9.2, if X or Y has a value that is an infinity or a NaN, the result shall be consistent with the general rules in 6.1 and 6.2 of the IEEE standard. For example, the result for an infinity shall be constructed as the limiting case of the result with a value of arbitrarily large magnitude, if such a limit exists.

14.10.1 IEEE_CLASS (X)

Description. IEEE class function.

Class. Elemental function.

Argument. X shall be of type real.

Restriction. IEEE_CLASS(X) shall not be invoked if IEEE_SUPPORT_DATATYPE(X) has the value false.

Result Characteristics. TYPE(IEEE_CLASS_TYPE).

Result Value. The result value shall be IEEE_SIGNALING_NAN or IEEE_QUIET_NAN if IEEE_SUPPORT_NAN(X) has the value true and the value of X is a signaling or quiet NaN, respectively. The result value shall be IEEE_NEGATIVE_INF or IEEE_POSITIVE_INF if IEEE_SUPPORT_INF(X) has the value true and the value of X is negative or positive infinity, respectively. The result value shall be IEEE_NEGATIVE_DENORMAL or IEEE_POSITIVE_DENORMAL if IEEE_SUPPORT_DENORMAL(X) has the value true and the value of X is a negative or positive denormalized value, respectively. The result value shall be IEEE_NEGATIVE_NORMAL, IEEE_NEGATIVE_ZERO, IEEE_POSITIVE_ZERO, or IEEE_POSITIVE_NORMAL if value of X is negative normal, negative zero, positive zero, or positive normal, respectively. Otherwise, the result value shall be IEEE_OTHER_VALUE.

Example. IEEE_CLASS(-1.0) has the value IEEE_NEGATIVE_NORMAL.

NOTE 14.10

> The result value IEEE_OTHER_VALUE is needed for implementing the module on systems which are basically IEEE, but do not implement all of it. It might be needed, for example, if an unformatted file were written in a program executing with gradual underflow enabled and read with it disabled.

14.10.2 IEEE_COPY_SIGN (X, Y)

Description. IEEE copysign function.

Class. Elemental function.

Arguments. The arguments shall be of type real.

Restriction. IEEE_COPY_SIGN(X,Y) shall not be invoked if IEEE_SUPPORT_DATATYPE(X) or IEEE_SUPPORT_DATATYPE(Y) has the value false.

Result Characteristics. Same as X.

Result Value. The result has the value of X with the sign of Y. This is true even for IEEE special values, such as a NaN or an infinity (on processors supporting such values).

Example. The value of IEEE_COPY_SIGN(X,1.0) is ABS(X) even when X is NaN.

14.10.3 IEEE_GET_FLAG (FLAG, FLAG_VALUE)

Description. Get an exception flag.

Class. Elemental subroutine.

Arguments.

FLAG	shall be of type TYPE(IEEE_FLAG_TYPE). It specifies the IEEE flag to be obtained.

FLAG_VALUE shall be of type default logical. It is an INTENT(OUT) argument. If the value of FLAG is IEEE_INVALID, IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_UNDERFLOW, or IEEE_INEXACT, the result value is true if the corresponding exception flag is signaling and is false otherwise.

Example. Following CALL IEEE_GET_FLAG(IEEE_OVERFLOW,FLAG_VALUE), FLAG_VALUE is true if the IEEE_OVERFLOW flag is signaling and is false if it is quiet.

14.10.4 IEEE_GET_HALTING_MODE (FLAG, HALTING)

Description. Get halting mode for an exception.

Class. Elemental subroutine.

Arguments.

FLAG shall be of type TYPE(IEEE_FLAG_TYPE). It specifies the IEEE flag. It shall have one of the values IEEE_INVALID, IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_UNDERFLOW, or IEEE_INEXACT.

HALTING shall be of type default logical. It is of INTENT(OUT). The value is true if the exception specified by FLAG will cause halting. Otherwise, the value is false.

Example. To store the halting mode for IEEE_OVERFLOW, do a calculation without halting, and restore the halting mode later:

```
USE, INTRINSIC :: IEEE_ARITHMETIC
LOGICAL HALTING
 ...
CALL IEEE_GET_HALTING_MODE(IEEE_OVERFLOW,HALTING) ! Store halting mode
CALL IEEE_SET_HALTING_MODE(IEEE_OVERFLOW,.FALSE.) ! No halting
 ...! calculation without halting
CALL IEEE_SET_HALTING_MODE(IEEE_OVERFLOW,HALTING) ! Restore halting mode
```

14.10.5 IEEE_GET_ROUNDING_MODE (ROUND_VALUE)

Description. Get the current IEEE rounding mode.

Class. Subroutine.

Argument. ROUND_VALUE shall be scalar of type TYPE(IEEE_ROUND_TYPE). It is an INTENT(OUT) argument and returns the floating point rounding mode, with value IEEE_NEAREST, IEEE_TO_ZERO, IEEE_UP, or IEEE_DOWN if one of the IEEE modes is in operation and IEEE_OTHER otherwise.

Example. To store the rounding mode, do a calculation with round to nearest, and restore the rounding mode later:

```
USE, INTRINSIC :: IEEE_ARITHMETIC
TYPE(IEEE_ROUND_TYPE) ROUND_VALUE
 ...
CALL IEEE_GET_ROUNDING_MODE(ROUND_VALUE) ! Store the rounding mode
```

```
CALL IEEE_SET_ROUNDING_MODE(IEEE_NEAREST)
 ... ! calculation with round to nearest
CALL IEEE_SET_ROUNDING_MODE(ROUND_VALUE) ! Restore the rounding mode
```

14.10.6 IEEE_GET_STATUS (STATUS_VALUE)

Description. Get the current value of the floating point status (14.6).

Class. Subroutine.

Argument. STATUS_VALUE shall be scalar of type TYPE(IEEE_STATUS_TYPE). It is an INTENT(OUT) argument and returns the floating point status.

Example. To store all the exception flags, do a calculation involving exception handling, and restore them later:

```
USE, INTRINSIC :: IEEE_ARITHMETIC
TYPE(IEEE_STATUS_TYPE) STATUS_VALUE
 ...
CALL IEEE_GET_STATUS(STATUS_VALUE) ! Get the flags
CALL IEEE_SET_FLAG(IEEE_ALL,.FALSE.) ! Set the flags quiet.
 ... ! calculation involving exception handling
CALL IEEE_SET_STATUS(STATUS_VALUE) ! Restore the flags
```

14.10.7 IEEE_GET_UNDERFLOW_MODE (GRADUAL)

Description. Get the current underflow mode (14.4).

Class. Subroutine.

Argument. GRADUAL shall be of type default logical. It is an INTENT(OUT) argument. The value is true if the current underflow mode is gradual underflow, and false if the current underflow mode is abrupt underflow.

Restriction. IEEE_GET_UNDERFLOW_MODE shall not be invoked unless IEEE_SUPPORT_UNDERFLOW_CONTROL(X) is true for some X.

Example. After CALL IEEE_SET_UNDERFLOW_MODE(.FALSE.), a subsequent CALL IEEE_GET_UNDERFLOW_MODE(GRADUAL) will set GRADUAL to false.

14.10.8 IEEE_IS_FINITE (X)

Description. Determine if a value is finite.

Class. Elemental function.

Argument. X shall be of type real.

Restriction. IEEE_IS_FINITE(X) shall not be invoked if IEEE_SUPPORT_DATATYPE(X) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value true if the value of X is finite, that is, IEEE_CLASS(X) has one of the values IEEE_NEGATIVE_NORMAL, IEEE_NEGATIVE_DENORMAL, IEEE_NEGATIVE_ZERO, IEEE_POSITIVE_ZERO, IEEE_POSITIVE_DENORMAL, or IEEE_POSITIVE_NORMAL; otherwise, the result has the value false.

Example. IEEE_IS_FINITE(1.0) has the value true.

14.10.9 IEEE_IS_NAN (X)

Description. Determine if a value is IEEE Not-a-Number.

Class. Elemental function.

Argument. X shall be of type real.

Restriction. IEEE_IS_NAN(X) shall not be invoked if IEEE_SUPPORT_NAN(X) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value true if the value of X is an IEEE NaN; otherwise, it has the value false.

Example. IEEE_IS_NAN(SQRT(-1.0)) has the value true if IEEE_SUPPORT_SQRT(1.0) has the value true.

14.10.10 IEEE_IS_NEGATIVE (X)

Description. Determine if a value is negative.

Class. Elemental function.

Argument. X shall be of type real.

Restriction. IEEE_IS_NEGATIVE(X) shall not be invoked if IEEE_SUPPORT_DATATYPE(X) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value true if IEEE_CLASS(X) has one of the values IEEE_NEGATIVE_NORMAL, IEEE_NEGATIVE_DENORMAL, IEEE_NEGATIVE_ZERO or IEEE_NEGATIVE_INF; otherwise, the result has the value false.

Example. IEEE_IS_NEGATIVE(0.0)) has the value false.

14.10.11 IEEE_IS_NORMAL (X)

Description. Determine if a value is normal, that is, neither an infinity, a NaN, nor denormalized.

Class. Elemental function.

Argument. X shall be of type real.

Restriction. IEEE_IS_NORMAL(X) shall not be invoked if IEEE_SUPPORT_DATATYPE(X) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value true if IEEE_CLASS(X) has one of the values IEEE_NEGATIVE_NORMAL, IEEE_NEGATIVE_ZERO, IEEE_POSITIVE_ZERO or IEEE_POSITIVE_NORMAL; otherwise, the result has the value false.

Example. IEEE_IS_NORMAL(SQRT(-1.0)) has the value false if IEEE_SUPPORT_SQRT(1.0) has the value true.

14.10.12 IEEE_LOGB (X)

Description. Unbiased exponent in the IEEE floating point format.

Class. Elemental function.

Argument. X shall be of type real.

Restriction. IEEE_LOGB(X) shall not be invoked if IEEE_SUPPORT_DATATYPE(X) has the value false.

Result Characteristics. Same as X.

Result Value.

Case (i): If the value of X is neither zero, infinity, nor NaN, the result has the value of the unbiased exponent of X. Note: this value is equal to EXPONENT(X)-1.

Case (ii): If X==0, the result is -infinity if IEEE_SUPPORT_INF(X) is true and -HUGE(X) otherwise; IEEE_DIVIDE_BY_ZERO signals.

Example. IEEE_LOGB(-1.1) has the value 0.0.

14.10.13 IEEE_NEXT_AFTER (X, Y)

Description. Returns the next representable neighbor of X in the direction toward Y.

Class. Elemental function.

Arguments. The arguments shall be of type real.

Restriction. IEEE_NEXT_AFTER(X,Y) shall not be invoked if IEEE_SUPPORT_DATATYPE(X) or IEEE_SUPPORT_DATATYPE(Y) has the value false.

Result Characteristics. Same as X.

Result Value.

Case (i): If X == Y, the result is X and no exception is signaled.

Case (ii): If X /= Y, the result has the value of the next representable neighbor of X in the direction of Y. The neighbors of zero (of either sign) are both nonzero. IEEE_OVERFLOW is signaled when X is finite but IEEE_NEXT_AFTER(X,Y) is infinite; IEEE_UNDERFLOW is signaled when IEEE_NEXT_AFTER(X,Y) is denormalized; in both cases, IEEE_INEXACT signals.

Example. The value of IEEE_NEXT_AFTER(1.0,2.0) is 1.0+EPSILON(X).

14.10.14 IEEE_REM (X, Y)

Description. IEEE REM function.

Class. Elemental function.

Arguments. The arguments shall be of type real.

Restriction. IEEE_REM(X,Y) shall not be invoked if IEEE_SUPPORT_DATATYPE(X) or IEEE_SUPPORT_DATATYPE(Y) has the value false.

Result Characteristics. Real with the kind type parameter of whichever argument has the greater precision.

Result Value. The result value, regardless of the rounding mode, shall be exactly X - Y*N, where N is the integer nearest to the exact value X/Y; whenever |N - X/Y| = 1/2, N shall be even. If the result value is zero, the sign shall be that of X.

Examples. The value of IEEE_REM(4.0,3.0) is 1.0, the value of IEEE_REM(3.0,2.0) is -1.0, and the value of IEEE_REM(5.0,2.0) is 1.0.

14.10.15 IEEE_RINT (X)

Description. Round to an integer value according to the current rounding mode.

Class. Elemental function.

Argument. X shall be of type real.

Restriction. IEEE_RINT(X) shall not be invoked if IEEE_SUPPORT_DATATYPE(X) has the value false.

Result Characteristics. Same as X.

Result Value. The value of the result is the value of X rounded to an integer according to the current rounding mode. If the result has the value zero, the sign is that of X.

Examples. If the current rounding mode is round to nearest, the value of IEEE_RINT(1.1) is 1.0. If the current rounding mode is round up, the value of IEEE_RINT(1.1) is 2.0.

14.10.16 IEEE_SCALB (X, I)

Description. Returns $X \times 2^I$.

Class. Elemental function.

Arguments.

X shall be of type real.

I shall be of type integer.

Restriction. IEEE_SCALB(X) shall not be invoked if IEEE_SUPPORT_DATATYPE(X) has the value false.

Result Characteristics. Same as X.

Result Value.

Case (i): If $X \times 2^I$ is representable as a normal number, the result has this value.

Case (ii): If X is finite and $X \times 2^I$ is too large, the IEEE_OVERFLOW exception shall occur. If IEEE_SUPPORT_INF(X) is true, the result value is infinity with the sign of X; otherwise, the result value is SIGN(HUGE(X),X).

Case (iii): If $X \times 2^I$ is too small and there is loss of accuracy, the IEEE_UNDERFLOW exception shall occur. The result is the representable number having a magnitude nearest to $|2^I|$ and the same sign as X.

Case (iv): If X is infinite, the result is the same as X; no exception signals.

Example. The value of IEEE_SCALB(1.0,2) is 4.0.

14.10.17 IEEE_SELECTED_REAL_KIND ([P, R])

Description. Returns a value of the kind type parameter of an IEEE real data type with decimal precision of at least P digits and a decimal exponent range of at least R. For data objects of such a type, IEEE_SUPPORT_DATATYPE(X) has the value true.

Class. Transformational function.

Arguments. At least one argument shall be present.

P (optional) shall be scalar and of type integer.

R (optional) shall be scalar and of type integer.

Result Characteristics. Default integer scalar.

Result Value. The result has a value equal to a value of the kind type parameter of an IEEE real type with decimal precision, as returned by the function PRECISION, of at least P digits and a decimal exponent range, as returned by the function RANGE, of at least R, or if no such kind type parameter is available on the processor, the result is -1 if the precision is not available, -2 if the exponent range is not available, and -3 if neither is available. If more than one kind type parameter value meets the criteria, the value returned is the one with the smallest decimal precision, unless there are several such values, in which case the smallest of these kind values is returned.

Example. IEEE_SELECTED_REAL_KIND(6,30) has the value KIND(0.0) on a machine that supports IEEE single precision arithmetic for its default real approximation method.

14.10.18 IEEE_SET_FLAG (FLAG, FLAG_VALUE)

Description. Assign a value to an exception flag.

Class. Subroutine.

Arguments.

FLAG shall be a scalar or array of type TYPE(IEEE_FLAG_TYPE). If a value of FLAG is IEEE_INVALID, IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_UNDERFLOW, or IEEE_INEXACT, the corresponding exception flag is assigned a value. No two elements of FLAG shall have the same value.

FLAG_VALUE shall be a scalar or array of type default logical. It shall be conformable with FLAG. If an element has the value true, the corresponding flag is set to be signaling; otherwise, the flag is set to be quiet.

Example. CALL IEEE_SET_FLAG(IEEE_OVERFLOW,.TRUE.) sets the IEEE_OVERFLOW flag to be signaling.

14.10.19 IEEE_SET_HALTING_MODE (FLAG, HALTING)

Description. Controls continuation or halting after an exception.

Class. Subroutine.

Arguments.

FLAG	shall be a scalar or array of type TYPE(IEEE_FLAG_TYPE). It shall have only the values IEEE_INVALID, IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_UNDERFLOW, or IEEE_INEXACT. No two elements of FLAG shall have the same value.
HALTING	shall be a scalar or array of type default logical. It shall be conformable with FLAG. If an element has the value is true, the corresponding exception specified by FLAG will cause halting. Otherwise, execution will continue after this exception.

Restriction. IEEE_SET_HALTING_MODE(FLAG,HALTING) shall not be invoked if IEEE_SUPPORT_HALTING(FLAG) has the value false.

Example. CALL IEEE_SET_HALTING_MODE(IEEE_DIVIDE_BY_ZERO,.TRUE.) causes halting after a divide_by_zero exception.

NOTE 14.11

The initial halting mode is processor dependent. Halting is not precise and may occur some time after the exception has occurred.

14.10.20 IEEE_SET_ROUNDING_MODE (ROUND_VALUE)

Description. Set the current IEEE rounding mode.

Class. Subroutine.

Argument. ROUND_VALUE shall be scalar and of type TYPE(IEEE_ROUND_TYPE). It specifies the mode to be set.

Restriction. IEEE_SET_ROUNDING_MODE(ROUND_VALUE) shall not be invoked unless IEEE_SUPPORT_ROUNDING(ROUND_VALUE,X) is true for some X such that IEEE_SUPPORT_DATATYPE(X) is true.

Example. To store the rounding mode, do a calculation with round to nearest, and restore the rounding mode later:

```
USE, INTRINSIC :: IEEE_ARITHMETIC
TYPE(IEEE_ROUND_TYPE) ROUND_VALUE
 ...
CALL IEEE_GET_ROUNDING_MODE(ROUND_VALUE) ! Store the rounding mode
CALL IEEE_SET_ROUNDING_MODE(IEEE_NEAREST)
: ! calculation with round to nearest
CALL IEEE_SET_ROUNDING_MODE(ROUND_VALUE) ! Restore the rounding mode
```

14.10.21 IEEE_SET_STATUS (STATUS_VALUE)

Description. Restore the value of the floating point status (14.6).

Class. Subroutine.

Argument. STATUS_VALUE shall be scalar and of type TYPE(IEEE_STATUS_TYPE). Its value shall have been set in a previous invocation of IEEE_GET_STATUS.

Example. To store all the exceptions flags, do a calculation involving exception handling, and restore them later:

```
USE, INTRINSIC :: IEEE_EXCEPTIONS
TYPE(IEEE_STATUS_TYPE) STATUS_VALUE
 ...
CALL IEEE_GET_STATUS(STATUS_VALUE) ! Store the flags
CALL IEEE_SET_FLAGS(IEEE_ALL,.FALSE.) ! Set them quiet
 ... ! calculation involving exception handling
CALL IEEE_SET_STATUS(STATUS_VALUE) ! Restore the flags
```

NOTE 14.12

On some processors this may be a very time consuming process.

14.10.22 IEEE_SET_UNDERFLOW_MODE (GRADUAL)

Description. Set the current underflow mode.

Class. Subroutine.

Argument. GRADUAL shall be of type default logical. If it is true, the current underflow mode is set to gradual underflow. If it is false, the current underflow mode is set to abrupt underflow.

Restriction. IEEE_SET_UNDERFLOW_MODE shall not be invoked unless IEEE_SUPPORT_UNDERFLOW_CONTROL(X) is true for some X.

Example. To perform some calculations with abrupt underflow and then restore the previous mode:

```
USE,INTRINSIC :: IEEE_ARITHMETIC
LOGICAL SAVE_UNDERFLOW_MODE
...
CALL IEEE_GET_UNDERFLOW_MODE(SAVE_UNDERFLOW_MODE)
CALL IEEE_SET_UNDERFLOW_MODE(GRADUAL=.FALSE.)
... ! Perform some calculations with abrupt underflow
CALL IEEE_SET_UNDERFLOW_MODE(SAVE_UNDERFLOW_MODE)"
```

14.10.23 IEEE_SUPPORT_DATATYPE () or IEEE_SUPPORT_DATATYPE (X)

Description. Inquire whether the processor supports IEEE arithmetic.

Class. Inquiry function.

Argument. X shall be of type real. It may be a scalar or an array.

Result Characteristics. Default logical scalar.

Result Value. The result has the value true if the processor supports IEEE arithmetic for all reals (X absent) or for real variables of the same kind type parameter as X; otherwise, it has the value false. Here, support is as defined in the first paragraph of 14.8.

Example. If default reals are implemented as in the IEEE standard except that underflow values flush to zero instead of being denormal, IEEE_SUPPORT_DATATYPE(1.0) has the value true.

14.10.24 IEEE_SUPPORT_DENORMAL () or IEEE_SUPPORT_DENORMAL (X)

Description. Inquire whether the processor supports IEEE denormalized numbers.

Class. Inquiry function.

Argument. X shall be of type real. It may be a scalar or an array.

Result Characteristics. Default logical scalar.

Result Value.

Case (i): IEEE_SUPPORT_DENORMAL(X) has the value true if IEEE_SUPPORT_DATATYPE(X) has the value true and the processor supports arithmetic operations and assignments with denormalized numbers (biased exponent $e = 0$ and fraction $f \neq 0$, see section 3.2 of the IEEE standard) for real variables of the same kind type parameter as X; otherwise, it has the value false.

Case (ii): IEEE_SUPPORT_DENORMAL() has the value true if and only if IEEE_SUPPORT_DENORMAL(X) has the value true for all real X.

Example. IEEE_SUPPORT_DENORMAL(X) has the value true if the processor supports denormalized numbers for X.

NOTE 14.13

The denormalized numbers are not included in the 13.4 model for real numbers; they satisfy the inequality ABS(X) < TINY(X). They usually occur as a result of an arithmetic operation whose exact result is less than TINY(X). Such an operation causes IEEE_UNDERFLOW to signal unless the result is exact. IEEE_SUPPORT_DENORMAL(X) is false if the processor never returns a denormalized number as the result of an arithmetic operation.

14.10.25 IEEE_SUPPORT_DIVIDE () or IEEE_SUPPORT_DIVIDE (X)

Description. Inquire whether the processor supports divide with the accuracy specified by the IEEE standard.

Class. Inquiry function.

Argument. X shall be of type real. It may be a scalar or an array.

Result Characteristics. Default logical scalar.

Result Value.

Case (i): IEEE_SUPPORT_DIVIDE(X) has the value true if the processor supports divide with the accuracy specified by the IEEE standard for real variables of the same kind type parameter as X; otherwise, it has the value false.

Case (ii): IEEE_SUPPORT_DIVIDE() has the value true if and only if IEEE_SUPPORT_DIVIDE(X) has the value true for all real X.

Example. IEEE_SUPPORT_DIVIDE(X) has the value true if the processor supports IEEE divide for X.

14.10.26 IEEE_SUPPORT_FLAG (FLAG) or IEEE_SUPPORT_FLAG (FLAG, X)

Description. Inquire whether the processor supports an exception.

Class. Inquiry function.

Arguments.

FLAG	shall be scalar and of type TYPE(IEEE_FLAG_TYPE). Its value shall be one of IEEE_INVALID, IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_UNDERFLOW, or IEEE_INEXACT.
X	shall be of type real. It may be a scalar or an array.

Result Characteristics. Default logical scalar.

Result Value.

Case (i): IEEE_SUPPORT_FLAG(FLAG, X) has the value true if the processor supports detection of the specified exception f(u)1(9552u)1(9552u)1(9552u)1(9552u)1(9552u)1(9552u)1(9552 finities (positive and negative) for real variables of the same kind type parameter as X; otherwise, it has the value false.

Case (ii): IEEE_SUPPORT_INF() has the value true if and only if IEEE_SUPPORT_INF(X) has the value true for all real X.

Example. IEEE_SUPPORT_INF(X) has the value true if the processor supports IEEE infinities for X.

14.10.29 IEEE_SUPPORT_IO () or IEEE_SUPPORT_IO (X)

Description. Inquire whether the processor supports IEEE base conversion rounding during formatted input/output (9.4.5.13, 9.5.1.12, 10.6.1.2.6).

Class. Inquiry function.

Argument. X shall be of type real. It may be a scalar or an array.

Result Characteristics. Default logical scalar.

Result Value.

Case (i): IEEE_SUPPORT_IO(X) has the value true if the processor supports IEEE base conversion during formatted input/output (9.4.5.13, 9.5.1.12, 10.6.1.2.6) as described in the IEEE standard for the modes UP, DOWN, ZERO, and NEAREST for real variables of the same kind type parameter as X; otherwise, it has the value false.

Case (ii): IEEE_SUPPORT_IO() has the value true if and only if IEEE_SUPPORT_IO(X) has the value true for all real X.

Example. IEEE_SUPPORT_IO(X) has the value true if the processor supports IEEE base conversion for X.

14.10.30 IEEE_SUPPORT_NAN () or IEEE_SUPPORT_NAN (X)

Description. Inquire whether the processor supports the IEEE Not-a-Number facility.

Class. Inquiry function.

Argument. X shall be of type real. It may be a scalar or an array.

Result Characteristics. Default logical scalar.

Result Value.

Case (i): IEEE_SUPPORT_NAN(X) has the value true if the processor supports IEEE NaNs for real variables of the same kind type parameter as X; otherwise, it has the value false.

Case (ii): IEEE_SUPPORT_NAN() has the value true if and only if IEEE_SUPPORT_NAN(X) has the value true for all real X.

Example. IEEE_SUPPORT_NAN(X) has the value true if the processor supports IEEE NaNs for X.

14.10.31 IEEE_SUPPORT_ROUNDING (ROUND_VALUE) or IEEE_SUPPORT_ROUNDING (ROUND_VALUE, X)

Description. Inquire whether the processor supports a particular IEEE rounding mode.

Class. Inquiry function.

Arguments.

ROUND_VALUE shall be of type TYPE(IEEE_ROUND_TYPE).

X shall be of type real. It may be a scalar or an array.

Result Characteristics. Default logical scalar.

Result Value.

Case (i): IEEE_SUPPORT_ROUNDING(ROUND_VALUE, X) has the value true if the processor supports the rounding mode defined by ROUND_VALUE for real variables of the same kind type parameter as X; otherwise, it has the value false. Support includes the ability to change the mode by CALL IEEE_SET_ROUNDING_MODE(ROUND_VALUE).

Case (ii): IEEE_SUPPORT_ROUNDING(ROUND_VALUE) has the value true if and only if IEEE_SUPPORT_ROUNDING(ROUND_VALUE, X) has the value true for all real X.

Example. IEEE_SUPPORT_ROUNDING(IEEE_TO_ZERO) has the value true if the processor supports rounding to zero for all reals.

14.10.32 IEEE_SUPPORT_SQRT () or IEEE_SUPPORT_SQRT (X)

Description. Inquire whether the processor implements SQRT in accord with the IEEE standard.

Class. Inquiry function.

Argument. X shall be of type real. It may be a scalar or an array.

Result Characteristics. Default logical scalar.

Result Value.

Case (i): IEEE_SUPPORT_SQRT(X) has the value true if the processor implements SQRT in accord with the IEEE standard for real variables of the same kind type parameter as X; otherwise, it has the value false.

Case (ii): IEEE_SUPPORT_SQRT() has the value true if and only if IEEE_SUPPORT_SQRT(X) has the value true for all real X.

Example. IEEE_SUPPORT_SQRT(X) has the value true if the processor implements SQRT(X) in accord with the IEEE standard. In this case, SQRT(-0.0) has the value -0.0.

14.10.33 IEEE_SUPPORT_STANDARD () or IEEE_SUPPORT_STANDARD (X)

Description. Inquire whether the processor supports all the IEEE facilities defined in this standard.

Class. Inquiry function.

Argument. X shall be of type real. It may be a scalar or an array.

Result Characteristics. Default logical scalar.

Result Value.

Case (i): IEEE_SUPPORT_STANDARD(X) has the value true if the results of all the functions IEEE_SUPPORT_DATATYPE(X), IEEE_SUPPORT_DENORMAL(X), IEEE_SUPPORT_DIVIDE(X), IEEE_SUPPORT_FLAG(FLAG,X) for valid FLAG, IEEE_SUPPORT_HALTING(FLAG) for valid FLAG, IEEE_SUPPORT_INF(X), IEEE_SUPPORT_NAN(X), IEEE_SUPPORT_ROUNDING (ROUND_VALUE,X) for valid ROUND_VALUE, and IEEE_SUPPORT_SQRT (X) are all true; otherwise, the result has the value false.

Case (ii): IEEE_SUPPORT_STANDARD() has the value true if and only if IEEE_SUPPORT_STANDARD(X) has the value true for all real X.

Example. IEEE_SUPPORT_STANDARD() has the value false if the processor supports both IEEE and non-IEEE kinds of reals.

14.10.34 IEEE_SUPPORT_UNDERFLOW_CONTROL() or IEEE_SUPPORT_UNDERFLOW_CONTROL(X)

Description. Inquire whether the procedure supports the ability to control the underflow mode during program execution.

Class. Inquiry function.

Argument. X shall be of type real. It may be a scalar or an array.

Result Characteristics. Default logical scalar.

Result Value.

Case (i): IEEE_SUPPORT_UNDERFLOW_CONTROL(X) has the value true if the processor supports control of the underflow mode for floating-point calculations with the same type as X, and false otherwise.

Case (ii): IEEE_SUPPORT_UNDERFLOW_CONTROL() has the value true if the processor supports control of the underflow mode for all floating-point calculations, and false otherwise.

Example. IEEE_SUPPORT_UNDERFLOW_CONTROL(2.5) has the value true if the processor supports underflow mode control for calculations of type default real.

14.10.35 IEEE_UNORDERED (X, Y)

Description. IEEE unordered function. True if X or Y is a NaN, and false otherwise.

Class. Elemental function.

Arguments. The arguments shall be of type real.

Restriction. IEEE_UNORDERED(X,Y) shall not be invoked if IEEE_SUPPORT_DATATYPE(X) or IEEE_SUPPORT_DATATYPE(Y) has the value false.

Result Characteristics. Default logical.

Result Value. The result has the value true if X or Y is a NaN or both are NaNs; otherwise, it has the value false.

Example. IEEE_UNORDERED(0.0,SQRT(-1.0)) has the value true if IEEE_SUPPORT_SQRT(1.0) has the value true.

14.10.36 IEEE_VALUE (X, CLASS)

Description. Generate an IEEE value.

Class. Elemental function.

Arguments.

X	shall be of type real.
CLASS	shall be of type TYPE(IEEE_CLASS_TYPE). The value is permitted to be: IEEE_SIGNALING_NAN or IEEE_QUIET_NAN if IEEE_SUPPORT_NAN(X) has the value true, IEEE_NEGATIVE_INF or IEEE_POSITIVE_INF if IEEE_SUPPORT_INF(X) has the value true, IEEE_NEGATIVE_DENORMAL or IEEE_POSITIVE_DENORMAL if IEEE_SUPPORT_DENORMAL(X) has the value true, IEEE_NEGATIVE_NORMAL, IEEE_NEGATIVE_ZERO, IEEE_POSITIVE_ZERO or IEEE_POSITIVE_NORMAL.

Restriction. IEEE_VALUE(X,CLASS) shall not be invoked if IEEE_SUPPORT_DATATYPE(X) has the value false.

Result Characteristics. Same as X.

Result Value. The result value is an IEEE value as specified by CLASS. Although in most cases the value is processor dependent, the value shall not vary between invocations for any particular X kind type parameter and CLASS value.

Example. IEEE_VALUE(1.0,IEEE_NEGATIVE_INF) has the value -infinity.

14.11 Examples

NOTE 14.14

```
MODULE DOT
! Module for dot product of two real arrays of rank 1.
! The caller needs to ensure that exceptions do not cause halting.
   USE, INTRINSIC :: IEEE_EXCEPTIONS
   LOGICAL :: MATRIX_ERROR = .FALSE.
   INTERFACE OPERATOR(.dot.)
      MODULE PROCEDURE MULT
   END INTERFACE
CONTAINS
   REAL FUNCTION MULT(A,B)
      REAL, INTENT(IN) :: A(:),B(:)
      INTEGER I
      LOGICAL OVERFLOW
      IF (SIZE(A)/=SIZE(B)) THEN
         MATRIX_ERROR = .TRUE.
         RETURN
      END IF
! The processor ensures that IEEE_OVERFLOW is quiet
      MULT = 0.0
      DO I = 1, SIZE(A)
```

NOTE 14.14 (cont.)

```
        MULT = MULT + A(I)*B(I)
     END DO
     CALL IEEE_GET_FLAG(IEEE_OVERFLOW,OVERFLOW)
     IF (OVERFLOW) MATRIX_ERROR = .TRUE.
  END FUNCTION MULT
END MODULE DOT
```

This module provides the dot product of two real arrays of rank 1. If the sizes of the arrays are different, an immediate return occurs with MATRIX_ERROR true. If overflow occurs during the actual calculation, the IEEE_OVERFLOW flag will signal and MATRIX_ERROR will be true.

NOTE 14.15

```
USE, INTRINSIC :: IEEE_EXCEPTIONS
USE, INTRINSIC :: IEEE_FEATURES, ONLY: IEEE_INVALID_FLAG
! The other exceptions of IEEE_USUAL (IEEE_OVERFLOW and
! IEEE_DIVIDE_BY_ZERO) are always available with IEEE_EXCEPTIONS
TYPE(IEEE_STATUS_TYPE) STATUS_VALUE
LOGICAL, DIMENSION(3) :: FLAG_VALUE
 ...
CALL IEEE_GET_STATUS(STATUS_VALUE)
CALL IEEE_SET_HALTING_MODE(IEEE_USUAL,.FALSE.) ! Needed in case the
!                 default on the processor is to halt on exceptions
CALL IEEE_SET_FLAG(IEEE_USUAL,.FALSE.)
! First try the "fast" algorithm for inverting a matrix:
MATRIX1 = FAST_INV(MATRIX) ! This shall not alter MATRIX.
CALL IEEE_GET_FLAG(IEEE_USUAL,FLAG_VALUE)
IF (ANY(FLAG_VALUE)) THEN
! "Fast" algorithm failed; try "slow" one:
   CALL IEEE_SET_FLAG(IEEE_USUAL,.FALSE.)
   MATRIX1 = SLOW_INV(MATRIX)
   CALL IEEE_GET_FLAG(IEEE_USUAL,FLAG_VALUE)
   IF (ANY(FLAG_VALUE)) THEN
      WRITE (*, *) 'Cannot invert matrix'
      STOP
   END IF
END IF
CALL IEEE_SET_STATUS(STATUS_VALUE)
```

In this example, the function FAST_INV may cause a condition to signal. If it does, another try is made with SLOW_INV. If this still fails, a message is printed and the program stops. Note, also, that it is important to set the flags quiet before the second try. The state of all the flags is stored and restored.

NOTE 14.16

```
USE, INTRINSIC :: IEEE_EXCEPTIONS
LOGICAL FLAG_VALUE
 ...
CALL IEEE_SET_HALTING_MODE(IEEE_OVERFLOW,.FALSE.)
! First try a fast algorithm for inverting a matrix.
CALL IEEE_SET_FLAG(IEEE_OVERFLOW,.FALSE.)
DO K = 1, N
 ...
   CALL IEEE_GET_FLAG(IEEE_OVERFLOW,FLAG_VALUE)
   IF (FLAG_VALUE) EXIT
END DO
IF (FLAG_VALUE) THEN
! Alternative code which knows that K-1 steps have executed normally.
 ...
END IF
```

Here the code for matrix inversion is in line and the transfer is made more precise by adding extra tests of the flag.

NOTE 14.17

```
REAL FUNCTION HYPOT(X, Y)
! In rare circumstances this may lead to the signaling of IEEE_OVERFLOW
! The caller needs to ensure that exceptions do not cause halting.
   USE, INTRINSIC :: IEEE_ARITHMETIC
   USE, INTRINSIC :: IEEE_FEATURES, ONLY: IEEE_UNDERFLOW_FLAG
! IEEE_OVERFLOW is always available with IEEE_ARITHMETIC
   REAL X, Y
   REAL SCALED_X, SCALED_Y, SCALED_RESULT
   LOGICAL, DIMENSION(2) :: FLAGS
   TYPE(IEEE_FLAG_TYPE), PARAMETER, DIMENSION(2) :: &
         OUT_OF_RANGE = (/ IEEE_OVERFLOW, IEEE_UNDERFLOW /)
   INTRINSIC SQRT, ABS, EXPONENT, MAX, DIGITS, SCALE
! The processor clears the flags on entry
! Try a fast algorithm first
   HYPOT = SQRT( X**2 + Y**2 )
   CALL IEEE_GET_FLAG(OUT_OF_RANGE,FLAGS)
   IF ( ANY(FLAGS) ) THEN
      CALL IEEE_SET_FLAG(OUT_OF_RANGE,.FALSE.)
      IF ( X==0.0 .OR. Y==0.0 ) THEN
         HYPOT = ABS(X) + ABS(Y)
      ELSE IF ( 2*ABS(EXPONENT(X)-EXPONENT(Y)) > DIGITS(X)+1 ) THEN
         HYPOT = MAX( ABS(X), ABS(Y) )! one of X and Y can be ignored
```

NOTE 14.17 (cont.)

```
      ELSE ! scale so that ABS(X) is near 1
         SCALED_X = SCALE( X, -EXPONENT(X) )
         SCALED_Y = SCALE( Y, -EXPONENT(X) )
         SCALED_RESULT = SQRT( SCALED_X**2 + SCALED_Y**2 )
         HYPOT = SCALE( SCALED_RESULT, EXPONENT(X) ) ! may cause overflow
      END IF
   END IF
! The processor resets any flag that was signaling on entry
END FUNCTION HYPOT
```

An attempt is made to evaluate this function directly in the fastest possible way. This will work almost every time, but if an exception occurs during this fast computation, a safe but slower way evaluates the function. This slower evaluation might involve scaling and unscaling, and in (very rare) extreme cases this unscaling can cause overflow (after all, the true result might overflow if X and Y are both near the overflow limit). If the IEEE_OVERFLOW or IEEE_UNDERFLOW flag is signaling on entry, it is reset on return by the processor, so that earlier exceptions are not lost.

Section 15: Interoperability with C

Fortran provides a means of referencing procedures that are defined by means of the C programming language or procedures that can be described by C prototypes as defined in 6.7.5.3 of the C standard, even if they are not actually defined by means of C. Conversely, there is a means of specifying that a procedure defined by a Fortran subprogram can be referenced from a function defined by means of C. In addition, there is a means of declaring global variables that are linked with C variables that have external linkage as defined in 6.2.2 of the C standard.

The ISO_C_BINDING module provides access to named constants that represent kind type parameters of data representations compatible with C types. Fortran also provides facilities for defining derived types (4.5) and enumerations (4.6) that correspond to C types.

15.1 The ISO_C_BINDING intrinsic module

The processor shall provide the intrinsic module ISO_C_BINDING. This module shall make accessible the following entities: named constants with the names listed in the second column of Table 15.2 and the first column of Table 15.1, the procedures specified in 15.1.2, C_PTR, C_FUNPTR, C_NULL_PTR, and C_NULL_FUNPTR. A processor may provide other public entities in the ISO_C_BINDING intrinsic module in addition to those listed here.

NOTE 15.1

To avoid potential name conflicts with program entities, it is recommended that a program use the ONLY option in any USE statement that accesses the ISO_C_BINDING intrinsic module.

15.1.1 Named constants and derived types in the module

The entities listed in the second column of Table 15.2, shall be named constants of type default integer.

The value of C_INT shall be a valid value for an integer kind parameter on the processor. The values of C_SHORT, C_LONG, C_LONG_LONG, C_SIGNED_CHAR, C_SIZE_T, C_INT8_T, C_INT16_T, C_-INT32_T, C_INT64_T, C_INT_LEAST8_T, C_INT_LEAST16_T, C_INT_LEAST32_T, C_INT_LEAST64-_T, C_INT_FAST8_T, C_INT_FAST16_T, C_INT_FAST32_T, C_INT_FAST64_T, C_INTMAX_T, and C_-INTPTR_T shall each be a valid value for an integer kind type parameter on the processor or shall be -1 if the companion C processor defines the corresponding C type and there is no interoperating Fortran processor kind or -2 if the C processor does not define the corresponding C type.

The values of C_FLOAT, C_DOUBLE, and C_LONG_DOUBLE shall each be a valid value for a real kind type parameter on the processor or shall be -1 if the C processor's type does not have a precision equal to the precision of any of the Fortran processor's real kinds, -2 if the C processor's type does not have a range equal to the range of any of the Fortran processor's real kinds, -3 if the C processor's type has neither the precision nor range of any of the Fortran processor's real kinds, and equal to -4 if there is no interoperating Fortran processor kind for other reasons. The values of C_COMPLEX, C_DOUBLE_COMPLEX, and C_LONG_DOUBLE_COMPLEX shall be the same as those of C_FLOAT, C_DOUBLE, and C_LONG_DOUBLE, respectively.

NOTE 15.2

If the C processor supports more than one variety of float, double or long double, the Fortran processor may find it helpful to select from among more than one ISO_C_BINDING module by a processor dependent means.

The value of C_BOOL shall be a valid value for a logical kind parameter on the processor or shall be -1.

The value of C_CHAR shall be a valid value for a character kind type parameter on the processor or shall be -1. The value of C_CHAR is known as the **C character kind**.

The following entities shall be named constants of type character with a length parameter of one. The kind parameter value shall be equal to the value of C_CHAR unless C_CHAR $= -1$, in which case the kind parameter value shall be the same as for default kind. The values of these constants are specified in Table 15.1. In the case that C_CHAR $\neq -1$ the value is specified using C syntax. The semantics of these values are explained in 5.2.1 and 5.2.2 of the C standard.

Table 15.1: **Names of C characters with special semantics**

Name	C definition	Value	
		C_CHAR $= -1$	C_CHAR $\neq -1$
C_NULL_CHAR	null character	CHAR(0)	`'\0'`
C_ALERT	alert	ACHAR(7)	`'\a'`
C_BACKSPACE	backspace	ACHAR(8)	`'\b'`
C_FORM_FEED	form feed	ACHAR(12)	`'\f'`
C_NEW_LINE	new line	ACHAR(10)	`'\n'`
C_CARRIAGE_RETURN	carriage return	ACHAR(13)	`'\r'`
C_HORIZONTAL_TAB	horizontal tab	ACHAR(9)	`'\t'`
C_VERTICAL_TAB	vertical tab	ACHAR(11)	`'\v'`

NOTE 15.3

The value of NEW_LINE(C_NEW_LINE) is C_NEW_LINE (13.7.85).

The entities C_PTR and C_FUNPTR are described in 15.2.2.

The entity C_NULL_PTR shall be a named constant of type C_PTR. The value of C_NULL_PTR shall be the same as the value NULL in C. The entity C_NULL_FUNPTR shall be a named constant of type C_FUNPTR. The value of C_NULL_FUNPTR shall be that of a null pointer to a function in C.

15.1.2 Procedures in the module

In the detailed descriptions below, procedure names are generic and not specific.

A C procedure argument is often defined in terms of a **C address**. The C_LOC and C_FUNLOC functions are provided so that Fortran applications can determine the appropriate value to use with C facilities. The C_ASSOCIATED function is provided so that Fortran programs can compare C addresses. The C_F_POINTER and C_F_PROCPOINTER subroutines provide a means of associating a Fortran pointer with the target of a C pointer.

15.1.2.1 C_ASSOCIATED (C_PTR_1 [, C_PTR_2])

Description. Indicates the association status of C_PTR_1 or indicates whether C_PTR_1 and C_PTR_2 are associated with the same entity.

Class. Inquiry function.

Arguments.

C_PTR_1 shall be a scalar of type C_PTR or C_FUNPTR.

C_PTR_2 (optional) shall be a scalar of the same type as C_PTR_1.

Result Characteristics. Default logical scalar.

Result Value.

Case (i): If C_PTR_2 is absent, the result is false if C_PTR_1 is a C null pointer and true otherwise.

Case (ii): If C_PTR_2 is present, the result is false if C_PTR_1 is a C null pointer. Otherwise, the result is true if C_PTR_1 compares equal to C_PTR_2 in the sense of 6.3.2.3 and 6.5.9 of the C standard, and false otherwise.

NOTE 15.4

The following example illustrates the use of C_LOC and C_ASSOCIATED.

```
USE, INTRINSIC :: ISO_C_BINDING, ONLY: C_PTR, C_FLOAT, C_ASSOCIATED, C_LOC
INTERFACE
  SUBROUTINE FOO(GAMMA), BIND(C)
    IMPORT C_PTR
    TYPE(C_PTR), VALUE :: GAMMA
  END SUBROUTINE FOO
END INTERFACE
REAL(C_FLOAT), TARGET, DIMENSION(100) :: ALPHA
TYPE(C_PTR) :: BETA
...
IF (.NOT. C_ASSOCIATED(BETA)) THEN
  BETA = C_LOC(ALPHA)
ENDIF
CALL FOO(BETA)
```

15.1.2.2 C_F_POINTER (CPTR, FPTR [, SHAPE])

Description. Associates a data pointer with the target of a C pointer and specifies its shape.

Class. Subroutine.

Arguments.

CPTR shall be a scalar of type C_PTR. It is an INTENT(IN) argument. Its value shall be:

(1) The C address of an interoperable data entity, or

(2) The result of a reference to C_LOC with a noninteroperable argument.

The value of CPTR shall not be the C address of a Fortran variable that does not have the TARGET attribute.

FPTR shall be a pointer. It is an INTENT(OUT) argument.

(1) If the value of CPTR is the C address of an interoperable data entity, FPTR shall be a data pointer with type and type parameters interoperable with the type of the entity. In this case, FPTR becomes pointer associated with the target of CPTR. If FPTR is an array, its shape is specified by SHAPE and each lower bound is 1.

(2) If the value of CPTR is the result of a reference to C_LOC with a noninteroperable argument X, FPTR shall be a nonpolymorphic scalar pointer with the same type and type parameters as X. In this case, X or its target if it is a pointer shall not have been deallocated or have become undefined due to execution of a RETURN or END statement since the reference. FPTR becomes pointer associated with X or its target.

SHAPE (optional) shall be of type integer and rank one. It is an INTENT(IN) argument. SHAPE shall be present if and only if FPTR is an array; its size shall be equal to the rank of FPTR.

15.1.2.3 C_F_PROCPOINTER (CPTR, FPTR)

Description. Associates a procedure pointer with the target of a C function pointer.

Class. Subroutine.

Arguments.

CPTR shall be a scalar of type C_FUNPTR. It is an INTENT(IN) argument. Its value shall be the C address of a procedure that is interoperable.

FPTR shall be a procedure pointer. It is an INTENT(OUT) argument. The interface for FPTR shall be interoperable with the prototype that describes the target of CPTR. FPTR becomes pointer associated with the target of CPTR.

NOTE 15.5

The term “target” in the descriptions of C_F_POINTER and C_F_PROCPOINTER denotes the entity referenced by a C pointer, as described in 6.2.5 of the C standard.

15.1.2.4 C_FUNLOC (X)

Description. Returns the C address of the argument.

Class. Inquiry function.

Argument. X shall either be a procedure that is interoperable, or a procedure pointer associated with an interoperable procedure.

Result Characteristics. Scalar of type C_FUNPTR.

Result Value.

The result value will be described using the result name CPTR. The result is determined as if C_FUNPTR were a derived type containing an implicit-interface procedure pointer component PX and the pointer assignment CPTR%PX => X were executed.

The result is a value that can be used as an actual CPTR argument in a call to C_F_PROC-

POINTER where FPTR has attributes that would allow the pointer assignment FPTR => X. Such a call to C_F_PROCPOINTER shall have the effect of the pointer assignment FPTR => X.

15.1.2.5 C_LOC (X)

Description. Returns the C address of the argument.

Class. Inquiry function.

Argument. X shall either

(1) have interoperable type and type parameters and be
  - (a) a variable that has the TARGET attribute and is interoperable,
  - (b) an allocated allocatable variable that has the TARGET attribute and is not an array of zero size, or
  - (c) an associated scalar pointer, or

(2) be a nonpolymorphic scalar, have no length type parameters, and be
  - (a) a nonallocatable, nonpointer variable that has the TARGET attribute,
  - (b) an allocated allocatable variable that has the TARGET attribute, or
  - (c) an associated pointer.

Result Characteristics. Scalar of type C_PTR.

Result Value.

The result value will be described using the result name CPTR.

(1) If X is a scalar data entity, the result is determined as if C_PTR were a derived type containing a scalar pointer component PX of the type and type parameters of X and the pointer assignment CPTR%PX => X were executed.

(2) If X is an array data entity, the result is determined as if C_PTR were a derived type containing a scalar pointer component PX of the type and type parameters of X and the pointer assignment of CPTR%PX to the first element of X were executed.

If X is a data entity that is interoperable or has interoperable type and type parameters, the result is the value that the C processor returns as the result of applying the unary "&" operator (as defined in the C standard, 6.5.3.2) to the target of CPTR

The result is a value that can be used as an actual CPTR argument in a call to C_F_POINTER where FPTR has attributes that would allow the pointer assignment FPTR => X. Such a call to C_F_POINTER shall have the effect of the pointer assignment FPTR => X.

NOTE 15.6

Where the actual argument is of noninteroperable type or type parameters, the result of C_LOC provides an opaque "handle" for it. In an actual implementation, this handle may be the C address of the argument; however, portable C functions should treat it as a void (generic) C pointer that cannot be dereferenced (6.5.3.2 in the C standard).

15.2 Interoperability between Fortran and C entities

The following subclauses define the conditions under which a Fortran entity is interoperable. If a Fortran entity is interoperable, an equivalent entity may be defined by means of C and the Fortran entity is said to be interoperable with the C entity. There does not have to be such an interoperating C entity.

NOTE 15.7

A Fortran entity can be interoperable with more than one C entity.

15.2.1 Interoperability of intrinsic types

Table 15.2 shows the interoperability between Fortran intrinsic types and C types. A Fortran intrinsic type with particular type parameter values is interoperable with a C type if the type and kind type parameter value are listed in the table on the same row as that C type; if the type is character, interoperability also requires that the length type parameter be omitted or be specified by an initialization expression whose value is one. A combination of Fortran type and type parameters that is interoperable with a C type listed in the table is also interoperable with any unqualified C type that is compatible with the listed C type.

The second column of the table refers to the named constants made accessible by the ISO_C_BINDING intrinsic module. If the value of any of these named constants is negative, there is no combination of Fortran type and type parameters interoperable with the C type shown in that row.

A combination of intrinsic type and type parameters is **interoperable** if it is interoperable with a C type.

Table 15.2: **Interoperability between Fortran and C types**

Fortran type	Named constant from the ISO_C_BINDING module (kind type parameter if value is positive)	C type
INTEGER	C_INT	int
	C_SHORT	short int
	C_LONG	long int
	C_LONG_LONG	long long int
	C_SIGNED_CHAR	signed char unsigned char
	C_SIZE_T	size_t
	C_INT8_T	int8_t
	C_INT16_T	int16_t
	C_INT32_T	int32_t
	C_INT64_T	int64_t
	C_INT_LEAST8_T	int_least8_t
	C_INT_LEAST16_T	int_least16_t
	C_INT_LEAST32_T	int_least32_t
	C_INT_LEAST64_T	int_least64_t
	C_INT_FAST8_T	int_fast8_t
	C_INT_FAST16_T	int_fast16_t
	C_INT_FAST32_T	int_fast32_t
	C_INT_FAST64_T	int_fast64_t
	C_INTMAX_T	intmax_t
	C_INTPTR_T	intptr_t
REAL	C_FLOAT	float
	C_DOUBLE	double

Interoperability between Fortran and C types (cont.)

Fortran type	Named constant from the ISO_C_BINDING module (kind type parameter if value is positive)	C type
	C_LONG_DOUBLE	long double
COMPLEX	C_FLOAT_COMPLEX	float _Complex
	C_DOUBLE_COMPLEX	double _Complex
	C_LONG_DOUBLE_COMPLEX	long double _Complex
LOGICAL	C_BOOL	_Bool
CHARACTER	C_CHAR	char
The above mentioned C types are defined in the C standard, clauses 6.2.5, 7.17, and 7.18.1.		

NOTE 15.8

For example, the type integer with a kind type parameter of C_SHORT is interoperable with the C type short or any C type derived (via typedef) from short.

NOTE 15.9

The C standard specifies that the representations for nonnegative signed integers are the same as the corresponding values of unsigned integers. Because Fortran does not provide direct support for unsigned kinds of integers, the ISO_C_BINDING module does not make accessible named constants for their kind type parameter values. Instead a user can use the signed kinds of integers to interoperate with the unsigned types and all their qualified versions as well. This has the potentially surprising side effect that the C type unsigned char is interoperable with the type integer with a kind type parameter of C_SIGNED_CHAR.

15.2.2 Interoperability with C pointer types

C_PTR and C_FUNPTR shall be a derived types with private components. C_PTR is interoperable with any C object pointer type. C_FUNPTR is interoperable with any C function pointer type.

NOTE 15.10

This implies that a C processor is required to have the same representation method for all C object pointer types and the same representation method for all C function pointer types if the C processor is to be the target of interoperability of a Fortran processor. The C standard does not impose this requirement.

NOTE 15.11

The function C_LOC can be used to return a value of type C_PTR that is the C address of an allocated allocatable variable. The function C_FUNLOC can be used to return a value of type C_FUNPTR that is the C address of a procedure. For C_LOC and C_FUNLOC the returned value is of an interoperable type and thus may be used in contexts where the procedure or allocatable variable is not directly allowed. For example, it could be passed as an actual argument to a C function.

Similarly, type C_FUNPTR or C_PTR can be used in a dummy argument or structure component and can have a value that is the C address of a procedure or allocatable variable, even in contexts where a procedure or allocatable variable is not directly allowed.

15.2.3 Interoperability of derived types and C struct types

A Fortran derived type is **interoperable** if it has the BIND attribute.

C1501 (R429) A derived type with the BIND attribute shall not be a SEQUENCE type.

C1502 (R429) A derived type with the BIND attribute shall not have type parameters.

C1503 (R429) A derived type with the BIND attribute shall not have the EXTENDS attribute.

C1504 (R429) A derived type with the BIND attribute shall not have a *type-bound-procedure-part*.

C1505 (R429) Each component of a derived type with the BIND attribute shall be a nonpointer, nonallocatable data component with interoperable type and type parameters.

NOTE 15.12

The syntax rules and their constraints require that a derived type that is interoperable have components that are all data objects that are interoperable. No component is permitted to be a procedure or allocatable, but a component of type C_FUNPTR or C_PTR may hold the C address of such an entity.

A Fortran derived type is interoperable with a C struct type if the derived-type definition of the Fortran type specifies BIND(C) (4.5.1), the Fortran derived type and the C struct type have the same number of components, and the components of the Fortran derived type have types and type parameters that are interoperable with the types of the corresponding components of the struct type. A component of a Fortran derived type and a component of a C struct type correspond if they are declared in the same relative position in their respective type definitions.

NOTE 15.13

The names of the corresponding components of the derived type and the C struct type need not be the same.

There is no Fortran type that is interoperable with a C struct type that contains a bit field or that contains a flexible array member. There is no Fortran type that is interoperable with a C union type.

NOTE 15.14

For example, the C type myctype, declared below, is interoperable with the Fortran type myftype, declared below.

```
typedef struct {
  int m, n;
  float r;
} myctype
```

```
USE, INTRINSIC :: ISO_C_BINDING
TYPE, BIND(C) :: MYFTYPE
  INTEGER(C_INT) :: I, J
  REAL(C_FLOAT) :: S
END TYPE MYFTYPE
```

The names of the types and the names of the components are not significant for the purposes of determining whether a Fortran derived type is interoperable with a C struct type.

NOTE 15.15

The C standard requires the names and component names to be the same in order for the types to be compatible (C standard, clause 6.2.7). This is similar to Fortran's rule describing when sequence derived types are considered to be the same type. This rule was not extended to determine whether a Fortran derived type is interoperable with a C struct type because the case of identifiers is significant in C but not in Fortran.

15.2.4 Interoperability of scalar variables

A scalar Fortran variable is **interoperable** if its type and type parameters are interoperable and it has neither the pointer nor the allocatable attribute.

An interoperable scalar Fortran variable is interoperable with a scalar C entity if their types and type parameters are interoperable.

15.2.5 Interoperability of array variables

An array Fortran variable is **interoperable** if its type and type parameters are interoperable and it is of explicit shape or assumed size.

An explicit-shape or assumed-size array of rank r, with a shape of $\left[\begin{array}{ccc} e_1 & \ldots & e_r \end{array}\right]$ is interoperable with a C array if its size is nonzero and

(1) either

(a) the array is assumed size, and the C array does not specify a size, or

(b) the array is an explicit shape array, and the extent of the last dimension (e_r) is the same as the size of the C array, and

(2) either

(a) r is equal to one, and an element of the array is interoperable with an element of the C array, or

(b) r is greater than one, and an explicit-shape array with shape of $\left[\begin{array}{ccc} e_1 & \ldots & e_{r-1} \end{array}\right]$, with the same type and type parameters as the original array, is interoperable with a C array of a type equal to the element type of the original C array.

NOTE 15.16

An element of a multi-dimensional C array is an array type, so a Fortran array of rank one is not interoperable with a multidimensional C array.

NOTE 15.17

A polymorphic, allocatable, or pointer array is never interoperable. Such arrays are not explicit shape or assumed size.

NOTE 15.18

For example, a Fortran array declared as

```
INTEGER :: A(18, 3:7, *)
```

is interoperable with a C array declared as

```
int b[][5][18]
```

NOTE 15.19

The C programming language defines null-terminated strings, which are actually arrays of the C type char that have a C null character in them to indicate the last valid element. A Fortran array of type character with a kind type parameter equal to C_CHAR is interoperable with a C string.

Fortran's rules of sequence association (12.4.1.5) permit a character scalar actual argument to be associated with a dummy argument array. This makes it possible to argument associate a Fortran character string with a C string.

Note 15.23 has an example of interoperation between Fortran and C strings.

15.2.6 Interoperability of procedures and procedure interfaces

A Fortran procedure is **interoperable** if it has the BIND attribute, that is, if its interface is specified with a *proc-language-binding-spec*.

A Fortran procedure interface is interoperable with a C function prototype if

(1) the interface has the BIND attribute;

(2) either

  (a) the interface describes a function whose result variable is a scalar that is interoperable with the result of the prototype or

  (b) the interface describes a subroutine, and the prototype has a result type of void;

(3) the number of dummy arguments of the interface is equal to the number of formal parameters of the prototype;

(4) any dummy argument with the VALUE attribute is interoperable with the corresponding formal parameter of the prototype;

(5) any dummy argument without the VALUE attribute corresponds to a formal parameter of the prototype that is of a pointer type, and the dummy argument is interoperable with an entity of the referenced type (C standard, 6.2.5, 7.17, and 7.18.1) of the formal parameter; and

(6) the prototype does not have variable arguments as denoted by the ellipsis (...).

NOTE 15.20

The **referenced type** of a C pointer type is the C type of the object that the C pointer type points to. For example, the referenced type of the pointer type `int *` is `int`.

NOTE 15.21

The C language allows specification of a C function that can take a variable number of arguments (C standard, 7.15). This standard does not provide a mechanism for Fortran procedures to interoperate with such C functions.

A formal parameter of a C function prototype corresponds to a dummy argument of a Fortran interface if they are in the same relative positions in the C parameter list and the dummy argument list, respectively.

NOTE 15.22

For example, a Fortran procedure interface described by

```
INTERFACE
  FUNCTION FUNC(I, J, K, L, M), BIND(C)
    USE, INTRINSIC :: ISO_C_BINDING
```

NOTE 15.22 (cont.)

```
    INTEGER(C_SHORT) :: FUNC
    INTEGER(C_INT), VALUE :: I
    REAL(C_DOUBLE) :: J
    INTEGER(C_INT) :: K, L(10)
    TYPE(C_PTR), VALUE :: M
  END FUNCTION FUNC
END INTERFACE
```

is interoperable with the C function prototype

```
short func(int i, double *j, int *k, int l[10], void *m)
```

A C pointer may correspond to a Fortran dummy argument of type C_PTR or to a Fortran scalar that does not have the VALUE attribute. In the above example, the C pointers `j` and `k` correspond to the Fortran scalars `J` and `K`, respectively, and the C pointer `m` corresponds to the Fortran dummy argument `M` of type C_PTR.

NOTE 15.23

The interoperability of Fortran procedure interfaces with C function prototypes is only one part of invocation of a C function from Fortran. There are four pieces to consider in such an invocation: the procedure reference, the Fortran procedure interface, the C function prototype, and the C function. Conversely, the invocation of a Fortran procedure from C involves the function reference, the C function prototype, the Fortran procedure interface, and the Fortran procedure. In order to determine whether a reference is allowed, it is necessary to consider all four pieces.

For example, consider a C function that can be described by the C function prototype

```
void copy(char in[], char out[]);
```

Such a function may be invoked from Fortran as follows:

```
  USE, INTRINSIC :: ISO_C_BINDING, ONLY: C_CHAR, C_NULL_CHAR
  INTERFACE
    SUBROUTINE COPY(IN, OUT), BIND(C)
      IMPORT C_CHAR
      CHARACTER(KIND=C_CHAR), DIMENSION(*) :: IN, OUT
    END SUBROUTINE COPY
  END INTERFACE

  CHARACTER(LEN=10, KIND=C_CHAR) :: &
 &     DIGIT_STRING = C_CHAR_'123456789' // C_NULL_CHAR
  CHARACTER(KIND=C_CHAR) :: DIGIT_ARR(10)

  CALL COPY(DIGIT_STRING, DIGIT_ARR)
  PRINT '(1X, A1)', DIGIT_ARR(1:9)
  END
```

NOTE 15.23 (cont.)

The procedure reference has character string actual arguments. These correspond to character array dummy arguments in the procedure interface body as allowed by Fortran's rules of sequence association (12.4.1.5). Those array dummy arguments in the procedure interface are interoperable with the formal parameters of the C function prototype. The C function is not shown here, but is assumed to be compatible with the C function prototype.

15.3 Interoperation with C global variables

A C variable with external linkage may interoperate with a common block or with a variable declared in the scope of a module. The common block or variable shall be specified to have the BIND attribute.

At most one variable that is associated with a particular C variable with external linkage is permitted to be declared within a program. A variable shall not be initially defined by more than one processor.

If a common block is specified in a BIND statement, it shall be specified in a BIND statement with the same binding label in each scoping unit in which it is declared. A C variable with external linkage interoperates with a common block that has been specified in a BIND statement

(1) if the C variable is of a struct type and the variables that are members of the common block are interoperable with corresponding components of the struct type, or

(2) if the common block contains a single variable, and the variable is interoperable with the C variable.

There does not have to be an associated C entity for a Fortran entity with the BIND attribute.

NOTE 15.24

The following are examples of the usage of the BIND attribute for variables and for a common block. The Fortran variables, C_EXTERN and C2, interoperate with the C variables, c_extern and myVariable, respectively. The Fortran common blocks, COM and SINGLE, interoperate with the C variables, com and single, respectively.

```
MODULE LINK_TO_C_VARS
  USE, INTRINSIC :: ISO_C_BINDING
  INTEGER(C_INT), BIND(C) :: C_EXTERN
  INTEGER(C_LONG) :: C2
  BIND(C, NAME='myVariable') :: C2

  COMMON /COM/ R, S
  REAL(C_FLOAT) :: R, S, T
  BIND(C) :: /COM/, /SINGLE/
  COMMON /SINGLE/ T
END MODULE LINK_TO_C_VARS
int c_extern;
long myVariable;
struct {float r, s;} com;
float single;
```

15.3.1 Binding labels for common blocks and variables

The **binding label** of a variable or common block is a value of type default character that specifies the name by which the variable or common block is known to the companion processor.

If a variable or common block has the BIND attribute specified with a NAME= specifier, the binding label is the value of the expression specified for the NAME= specifier. The case of letters in the binding label is significant, but leading and trailing blanks are ignored. If a variable or common block has the BIND attribute specified without a NAME= specifier, the binding label is the same as the name of the entity using lower case letters.

The binding label of a C variable with external linkage is the same as the name of the C variable. A Fortran variable or common block with the BIND attribute that has the same binding label as a C variable with external linkage is associated with that variable.

15.4 Interoperation with C functions

A procedure that is interoperable may be defined either by means other than Fortran or by means of a Fortran subprogram, but not both.

If the procedure is defined by means other than Fortran, it shall

(1) be describable by a C prototype that is interoperable with the interface,

(2) have external linkage as defined by 6.2.2 of the C standard, and

(3) have the same binding label as the interface.

A reference to such a procedure causes the function described by the C prototype to be called as specified in the C standard.

A reference in C to a procedure that has the BIND attribute, has the same binding label, and is defined by means of Fortran, causes the Fortran procedure to be invoked.

A procedure defined by means of Fortran shall not invoke setjmp or longjmp (C standard, 7.13). If a procedure defined by means other than Fortran invokes setjmp or longjmp, that procedure shall not cause any procedure defined by means of Fortran to be invoked. A procedure defined by means of Fortran shall not be invoked as a signal handler (C standard, 7.14.1).

If a procedure defined by means of Fortran and a procedure defined by means other than Fortran perform input/output operations on the same external file, the results are processor dependent (9.4.3).

15.4.1 Binding labels for procedures

A **binding label** is a value of type default character that specifies the name by which a procedure with the BIND attribute is known to the companion processor.

If a procedure has the BIND attribute with the NAME= specifier, the procedure has a binding label whose value is that of the expression in the NAME= specifier. The case of letters in the binding label is significant, but leading and trailing blanks are ignored. If a procedure has the BIND attribute with no NAME= specifier, and the procedure is not a dummy procedure, then the binding label of the procedure is the same as the name of the procedure using lower case letters.

The binding label for a C function with external linkage is the same as the C function name.

NOTE 15.25

In the following sample, the binding label of C_SUB is "c_sub", and the binding label of C_FUNC is "C_funC".

```
SUBROUTINE C_SUB, BIND(C)
  ...
END SUBROUTINE C_SUB

INTEGER(C_INT) FUNCTION C_FUNC(), BIND(C, NAME="C_funC")
  USE, INTRINSIC :: ISO_C_BINDING
  ...
END FUNCTION C_FUNC
```

The C standard permits functions to have names that are not permitted as Fortran names; it also distinguishes between names that would be considered as the same name in Fortran. For example, a C name may begin with an underscore, and C names that differ in case are distinct names.

The specification of a binding label allows a program to use a Fortran name to refer to a procedure defined by a companion processor.

15.4.2 Exceptions and IEEE arithmetic procedures

A procedure defined by means other than Fortran shall not use signal (C standard, 7.14.1) to change the handling of any exception that is being handled by the Fortran processor.

A procedure defined by means other than Fortran shall not alter the floating point status (14.6 other than by setting an exception flag to signaling.

The values of the floating point exception flags on entry to a procedure defined by means other than Fortran are processor-dependent.

Section 16: Scope, association, and definition

Entities are identified by identifiers within a **scope** that is a program, a scoping unit, a construct, a single statement, or part of a statement.

- A **global identifier** has a scope of a program (2.2.1);
- A **local identifier** has a scope of a scoping unit (2.2);
- An identifier of a **construct entity** has a scope of a construct (7.4.3, 7.4.4, 8.1);
- An identifier of a **statement entity** has a scope of a statement or part of a statement (3.3).

An entity may be identified by

(1) A name (3.2.1),
(2) A statement label (3.2.4),
(3) An external input/output unit number (9.4),
(4) An identifier of a pending data transfer operation (9.5.1.8, 9.6),
(5) A generic identifier (12.3.2.1), or
(6) A binding label (15.4.1, 15.3.1).

By means of association, an entity may be referred to by the same identifier or a different identifier in a different scoping unit, or by a different identifier in the same scoping unit.

16.1 Scope of global identifiers

Program units, common blocks, external procedures, procedure binding labels, and variables that have the BIND attribute are **global entities** of a program. The name of a program unit, common block, or external procedure is a global identifier and shall not be the same as the name of any other such global entity in the same program, except that an intrinsic module may have the same name as another program unit, common block, or external procedure in the same program. A binding label of a global entity of the program is a global identifier and shall not be the same as the binding label of any other global entity of the program; nor shall it be the same as a name of any other global entity of the program that is not an intrinsic module, ignoring differences in case. A global entity of the program shall not be identified by more than one binding label.

NOTE 16.1

The name of a global entity may be the same as a binding label that identifies the same global entity.

NOTE 16.2

Of the various types of procedures, only external procedures have global names. An implementation may wish to assign global names to other entities in the Fortran program such as internal procedures, intrinsic procedures, procedures implementing intrinsic operators, procedures implementing input/output operations, etc. If this is done, it is the responsibility of the processor to ensure that none of these names conflicts with any of the names of the external procedures, with other globally named entities in a standard-conforming program, or with each other. For example,

NOTE 16.2 (cont.)

this might be done by including in each such added name a character that is not allowed in a standard-conforming name or by using such a character to combine a local designation with the global name of the program unit in which it appears.

External input/output units and pending data transfer operations are global entities.

16.2 Scope of local identifiers

Within a scoping unit, identifiers of entities in the following classes:

(1) Named variables that are not statement or construct entities (16.3), named constants, named constructs, statement functions, internal procedures, module procedures, dummy procedures, intrinsic procedures, abstract interfaces, generic interfaces, derived types, namelist groups, external procedures accessed via USE, and statement labels,

(2) Type parameters, components, and type-bound procedure bindings, in a separate class for each type, and

(3) Argument keywords, in a separate class for each procedure with an explicit interface

are local identifiers in that scoping unit.

Within a scoping unit, a local identifier of an entity of class (1) shall not be the same as a global identifier used in that scoping unit unless the global identifier

(1) is used only as the *use-name* of a *rename* in a USE statement,

(2) is a common block name (16.2.1),

(3) is an external procedure name that is also a generic name, or

(4) is an external function name and the scoping unit is its defining subprogram (16.2.2).

Within a scoping unit, a local identifier of one class shall not be the same as another local identifier of the same class, except that a generic name may be the same as the name of a procedure as explained in 12.3.2.1 or the same as the name of a derived type (4.5.9). A local identifier of one class may be the same as a local identifier of another class.

NOTE 16.3

An intrinsic procedure is inaccessible by its own name in a scoping unit that uses the same name as a local identifier of class (1) for a different entity. For example, in the program fragment

```
SUBROUTINE SUB
   ...
   A = SIN (K)
   ...
CONTAINS
   FUNCTION SIN (X)
      ...
   END FUNCTION SIN
END SUBROUTINE SUB
```

any reference to function SIN in subroutine SUB refers to the internal function SIN, not to the intrinsic function of the same name.

A local identifier identifies an entity in a scoping unit and may be used to identify an entity in another scoping unit except in the following cases:

(1) The name that appears as a *subroutine-name* in a *subroutine-stmt* has limited use within the scope established by the *subroutine-stmt*. It can be used to identify recursive references of the subroutine or to identify a common block (the latter is possible only for internal and module subroutines).

(2) The name that appears as a *function-name* in a *function-stmt* has limited use within the scope established by that *function-stmt*. It can be used to identify the result variable, to identify recursive references of the function, or to identify a common block (the latter is possible only for internal and module functions).

(3) The name that appears as an *entry-name* in an *entry-stmt* has limited use within the scope of the subprogram in which the *entry-stmt* appears. It can be used to identify the result variable if the subprogram is a function, to identify recursive references, or to identify a common block (the latter is possible only if the *entry-stmt* is in a module subprogram).

16.2.1 Local identifiers that are the same as common block names

A name that identifies a common block in a scoping unit shall not be used to identify a constant or an intrinsic procedure in that scoping unit. If a local identifier is also the name of a common block, the appearance of that name in any context other than as a common block name in a COMMON or SAVE statement is an appearance of the local identifier.

NOTE 16.4

An intrinsic procedure name may be a common block name in a scoping unit that does not reference the intrinsic procedure.

16.2.2 Function results

For each FUNCTION statement or ENTRY statement in a function subprogram, there is a result variable. If there is no RESULT clause, the result variable has the same name as the function being defined; otherwise, the result variable has the name specified in the RESULT clause.

16.2.3 Restrictions on generic declarations

This subclause contains the rules that shall be satisfied by every pair of specific procedures that have the same generic identifier within a scoping unit. If a generic procedure is accessed from a module, the rules apply to all the specific versions even if some of them are inaccessible by their specific names.

NOTE 16.5

In most scoping units, the possible sources of procedures with a particular generic identifier are the accessible interface blocks and the generic bindings other than names for the accessible objects in that scoping unit. In a type definition, they are the generic bindings, including those from a parent type.

Two dummy arguments are **distinguishable** if neither is a subroutine and neither is TKR compatible (5.1.1.2) with the other.

Within a scoping unit, if two procedures have the same generic operator and the same number of arguments or both define assignment, one shall have a dummy argument that corresponds by position in the argument list to a dummy argument of the other that is distinguishable with it.

Within a scoping unit, if two procedures have the same *dtio-generic-spec* (12.3.2.1), their `dtv` arguments shall be type incompatible or have different kind type parameters.

Within a scoping unit, two procedures that have the same generic name shall both be subroutines or both be functions, and

(1) there is a non-passed-object dummy data object in one or the other of them such that

  (a) the number of dummy data objects in one that are nonoptional, are not passed-object, and with which that dummy data object is TKR compatible, possibly including that dummy data object itself,

  exceeds

  (b) the number of non-passed-object dummy data objects, both optional and nonoptional, in the other that are not distinguishable with that dummy data object;

(2) both have passed-object dummy arguments and the passed-object dummy arguments are distinguishable; or

(3) at least one of them shall have both

  (a) A nonoptional non-passed-object dummy argument at an effective position such that either the other procedure has no dummy argument at that effective position or the dummy argument at that position is distinguishable with it; and

  (b) A nonoptional non-passed-object dummy argument whose name is such that either the other procedure has no dummy argument with that name or the dummy argument with that name is distinguishable with it.

  Further, the dummy argument that disambiguates by position shall either be the same as or occur earlier in the argument list than the one that disambiguates by name.

The **effective position** of a dummy argument is its position in the argument list after any passed-object dummy argument has been removed.

Within a scoping unit, if a generic name is the same as the name of a generic intrinsic procedure, the generic intrinsic procedure is not accessible if the procedures in the interface and the intrinsic procedure are not all functions or not all subroutines. If a generic invocation applies to both a specific procedure from an interface and an accessible generic intrinsic procedure, it is the specific procedure from the interface that is referenced.

NOTE 16.6

An extensive explanation of the application of these rules is in C.11.2.

16.2.4 Components, type parameters, and bindings

A component name has the scope of its derived-type definition. Outside the type definition, it may appear only within a designator of a component of a structure of that type or as a component keyword in a structure constructor for that type.

A type parameter name has the scope of its derived-type definition. Outside the derived-type definition, it may appear only as a type parameter keyword in a *derived-type-spec* for the type or as the *type-param-name* of a *type-param-inquiry*.

The binding name (4.5.4)of a type-bound procedure has the scope of its derived-type definition. Outside of the derived-type definition, it may appear only as the *binding-name* in a procedure reference.

A generic binding for which the *generic-spec* is not a *generic-name* has a scope that consists of all scoping units in which an entity of the type is accessible.

A component name or binding name may appear only in scoping units in which it is accessible.

The accessibility of components and bindings is specified in 4.5.3.6 and 4.5.4.

16.2.5 Argument keywords

As an argument keyword, a dummy argument name in an internal procedure, module procedure, or an interface body has a scope of the scoping unit of the host of the procedure or interface body. It may appear only in a procedure reference for the procedure of which it is a dummy argument. If the procedure or interface body is accessible in another scoping unit by use association or host association (16.4.1.2, 16.4.1.3), the argument keyword is accessible for procedure references for that procedure in that scoping unit.

A dummy argument name in an intrinsic procedure has a scope as an argument keyword of the scoping unit in which the reference to the procedure occurs. As an argument keyword, it may appear only in a procedure reference for the procedure of which it is a dummy argument.

16.3 Statement and construct entities

A variable that appears as a *data-i-do-variable* in a DATA statement or an *ac-do-variable* in an array constructor, as a dummy argument in a statement function statement, or as an *index-name* in a FORALL statement is a statement entity. A variable that appears as an *index-name* in a FORALL construct or an *associate-name* in a SELECT TYPE or ASSOCIATE construct is a construct entity.

The name of a *data-i-do-variable* in a DATA statement or an *ac-do-variable* in an array constructor has a scope of its *data-implied-do*. It is a scalar variable that has the type and type parameters that it would have if it were the name of a variable in the scoping unit that includes the DATA statement or array constructor, and this type shall be integer type; it has no other attributes. The appearance of a name as a *data-i-do-variable* of an implied-DO in a DATA statement or an *ac-do-variable* in an array constructor is not an implicit declaration of a variable whose scope is the scoping unit that contains the statement.

The name of a variable that appears as an *index-name* in a FORALL statement or FORALL construct has a scope of the statement or construct. It is a scalar variable that has the type and type parameters that it would have if it were the name of a variable in the scoping unit that includes the FORALL, and this type shall be integer type; it has no other attributes. The appearance of a name as an index-name in a FORALL statement or FORALL construct is not an implicit declaration of a variable whose scope is the scoping unit that contains the statement or construct.

The name of a variable that appears as a dummy argument in a statement function statement has a scope of the statement in which it appears. It is a scalar that has the type and type parameters that it would have if it were the name of a variable in the scoping unit that includes the statement function; it has no other attributes.

Except for a common block name or a scalar variable name, a global identifier or a local identifier of class (1) (16.2) in the scoping unit that contains a statement shall not be the name of a statement entity of that statement. Within the scope of a statement entity, another statement entity shall not have the same name.

If a global or local identifier accessible in the scoping unit of a statement is the same as the name of a statement entity in that statement, the name is interpreted within the scope of the statement entity as that of the statement entity. Elsewhere in the scoping unit, including parts of the statement outside the scope of the statement entity, the name is interpreted as the global or local identifier.

Except for a common block name or a scalar variable name, a global identifier or a local identifier of class (1) (16.2) in the scoping unit of a FORALL statement or FORALL construct shall not be the same as any of its *index-name*s. An *index-name* of a contained FORALL statement or FORALL construct shall not be the same as an *index-name* of any of its containing FORALL constructs.

If a global or local identifier accessible in the scoping unit of a FORALL statement or FORALL construct is the same as the *index-name*, the name is interpreted within the scope of the FORALL statement or FORALL construct as that of the *index-name*. Elsewhere in the scoping unit, the name is interpreted as the global or local identifier.

The associate name of a SELECT TYPE construct has a separate scope for each block of the construct. Within each block, it has the declared type, dynamic type, type parameters, rank, and bounds specified in 8.1.5.2.

The associate names of an ASSOCIATE construct have the scope of the block. They have the declared type, dynamic type, type parameters, rank, and bounds specified in 8.1.4.2.

If a global or local identifier accessible in the scoping unit of a SELECT TYPE or ASSOCIATE construct is the same as an associate name, the name is interpreted within the blocks of the SELECT TYPE or ASSOCIATE construct as that of the associate name. Elsewhere in the scoping unit, the name is interpreted as the global or local identifier.

16.4 Association

Two entities may become associated by name association, pointer association, storage association, or inheritance association.

16.4.1 Name association

There are five forms of **name association**: argument association, use association, host association, linkage association, and construct association. Argument, use, and host association provide mechanisms by which entities known in one scoping unit may be accessed in another scoping unit.

16.4.1.1 Argument association

The rules governing argument association are given in Section 12. As explained in 12.4, execution of a procedure reference establishes an association between an actual argument and its corresponding dummy argument. Argument association may be sequence association (12.4.1.5).

The name of the dummy argument may be different from the name, if any, of its associated actual argument. The dummy argument name is the name by which the associated actual argument is known, and by which it may be accessed, in the referenced procedure.

NOTE 16.7

An actual argument may be a nameless data entity, such as an expression that is not simply a variable or constant.

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 invocation of the procedure.

16.4.1.2 Use association

Use association is the association of names in different scoping units specified by a USE statement. The rules governing use association are given in 11.2.1. They allow for renaming of entities being accessed. Use association allows access in one scoping unit to entities defined in another scoping unit; it remains in effect throughout the execution of the program.

16.4.1.3 Host association

An internal subprogram, a module subprogram, or a derived-type definition has access to the named entities from its host via **host association**. An interface body has access via host association to the named entities from its host that are made accessible by IMPORT statements in the interface body. The accessed entities are known by the same name and have the same attributes as in the host; they are named data objects, derived types, abstract interfaces, procedures, generic identifiers (12.3.2.1), and namelist groups.

If an entity that is accessed by use association has the same nongeneric name as a host entity, the host entity is inaccessible by that name. Within the scoping unit, a name that is declared to be an external procedure name by an *external-stmt*, *procedure-declaration-stmt*, or *interface-body*, or that appears as a *module-name* in a *use-stmt* is a global identifier; any entity of the host that has this as its nongeneric name is inaccessible by that name. A name that appears in the scoping unit as

(1) A *function-name* in a *stmt-function-stmt* or in an *entity-decl* in a *type-declaration-stmt*;
(2) An *object-name* in an *entity-decl* in a *type-declaration-stmt*, in a *pointer-stmt*, in a *save-stmt*, in an *allocatable-stmt*, or in a *target-stmt*;
(3) A *type-param-name* in a *derived-type-stmt*;
(4) A *named-constant* in a *named-constant-def* in a *parameter-stmt*;
(5) An *array-name* in a *dimension-stmt*;
(6) A *variable-name* in a *common-block-object* in a *common-stmt*;
(7) A *proc-pointer-name* in a *common-block-object* in a *common-stmt*;
(8) The name of a variable that is wholly or partially initialized in a *data-stmt*;
(9) The name of an object that is wholly or partially equivalenced in an *equivalence-stmt*;
(10) A *dummy-arg-name* in a *function-stmt*, in a *subroutine-stmt*, in an *entry-stmt*, or in a *stmt-function-stmt*;
(11) A *result-name* in a *function-stmt* or in an *entry-stmt*;
(12) The name of an entity declared by an interface body;
(13) An *intrinsic-procedure-name* in an *intrinsic-stmt*;
(14) A *namelist-group-name* in a *namelist-stmt*;
(15) A *generic-name* in a *generic-spec* in an *interface-stmt*; or
(16) The name of a named construct

is a local identifier in the scoping unit and any entity of the host that has this as its nongeneric name is inaccessible by that name by host association. If a scoping unit is the host of a derived-type definition or a subprogram, the name of the derived type or of any procedure defined by the subprogram is a local identifier in the scoping unit; any entity of the host that has this as its nongeneric name is inaccessible by that name. Local identifiers of a subprogram are not accessible to its host.

NOTE 16.8

A name that appears in an ASYNCHRONOUS or VOLATILE statement is not necessarily the name of a local variable. If a variable that is accessible via host association other than by an IMPORT statement is specified in an ASYNCHRONOUS or VOLATILE statement, that host variable is given the ASYNCHRONOUS or VOLATILE attribute in the scope of the current internal or module procedure.

If a host entity is inaccessible only because a local variable with the same name is wholly or partially initialized in a DATA statement, the local variable shall not be referenced or defined prior to the DATA statement.

If a derived-type name of a host is inaccessible, data entities of that type or subobjects of such data entities still can be accessible.

NOTE 16.9

An interface body accesses by host association only those entities made accessible by IMPORT statements.

If an external or dummy procedure with an implicit interface is accessed via host association, then it shall have the EXTERNAL attribute in the host scoping unit; if it is invoked as a function in the inner scoping unit, its type and type parameters shall be established in the host scoping unit. The type and type parameters of a function with the EXTERNAL attribute are established in a scoping unit if that scoping unit explicitly declares them, invokes the function, accesses the function from a module, or accesses the function from its host where its type and type parameters are established.

If an intrinsic procedure is accessed via host association, then it shall be established to be intrinsic in the host scoping unit. An intrinsic procedure is established to be intrinsic in a scoping unit if that scoping unit explicitly gives it the INTRINSIC attribute, invokes it as an intrinsic procedure, accesses it from a module, or accesses it from its host where it is established to be intrinsic.

NOTE 16.10

A host subprogram and an internal subprogram may contain the same and differing use-associated entities, as illustrated in the following example.

```
MODULE B; REAL BX, Q; INTEGER IX, JX; END MODULE B
MODULE C; REAL CX; END MODULE C
MODULE D; REAL DX, DY, DZ; END MODULE D
MODULE E; REAL EX, EY, EZ; END MODULE E
MODULE F; REAL FX; END MODULE F
MODULE G; USE F; REAL GX; END MODULE G
PROGRAM A
USE B; USE C; USE D
   ...
CONTAINS
   SUBROUTINE INNER_PROC (Q)
      USE C            ! Not needed
      USE B, ONLY: BX  ! Entities accessible are BX, IX, and JX
                       ! if no other IX or JX
                       ! is accessible to INNER_PROC
                       ! Q is local to INNER_PROC,
                       ! because Q is a dummy argument
      USE D, X => DX   ! Entities accessible are DX, DY, and DZ
                       ! X is local name for DX in INNER_PROC
                       ! X and DX denote same entity if no other
                       ! entity DX is local to INNER_PROC
      USE E, ONLY: EX  ! EX is accessible in INNER_PROC, not in program A
                       ! EY and EZ are not accessible in INNER_PROC
                       ! or in program A
      USE G            ! FX and GX are accessible in INNER_PROC
      ...
```

NOTE 16.10 (cont.)

```
   END SUBROUTINE INNER_PROC
END PROGRAM A
```

Because program A contains the statement

```
USE B
```

all of the entities in module B, except for Q, are accessible in INNER_PROC, even though INNER_PROC contains the statement

```
USE B, ONLY: BX
```

The USE statement with the ONLY keyword means that this particular statement brings in only the entity named, not that this is the only variable from the module accessible in this scoping unit.

NOTE 16.11

For more examples of host association, see section C.11.1.

16.4.1.4 Linkage association

Linkage association occurs between a module variable that has the BIND attribute and the C variable with which it interoperates, or between a Fortran common block and the C variable with which it interoperates (15.3). Such association remains in effect throughout the execution of the program.

16.4.1.5 Construct association

Execution of a SELECT TYPE statement establishes an association between the selector and the associate name of the construct. Execution of an ASSOCIATE statement establishes an association between each selector and the corresponding associate name of the construct.

If the selector is allocatable, it shall be allocated; the associate name is associated with the data object and does not have the ALLOCATABLE attribute.

If the selector has the POINTER attribute, it shall be associated; the associate name is associated with the target of the pointer and does not have the POINTER attribute.

If the selector is a variable other than an array section having a vector subscript, the association is with the data object specified by the selector; otherwise, the association is with the value of the selector expression, which is evaluated prior to execution of the block.

Each associate name remains associated with the corresponding selector throughout the execution of the executed block. Within the block, each selector is known by and may be accessed by the corresponding associate name. Upon termination of the construct, the association is terminated.

NOTE 16.12

The association between the associate name and a data object is established prior to execution of the block and is not affected by subsequent changes to variables that were used in subscripts or substring ranges in the *selector*.

16.4.2 Pointer association

Pointer association between a pointer and a target allows the target to be referenced by a reference to the pointer. At different times during the execution of a program, a pointer may be undefined, associated with different targets, or be disassociated. If a pointer is associated with a target, the definition status of the pointer is either defined or undefined, depending on the definition status of the target. If the pointer has deferred type parameters or shape, their values are assumed from the target. If the pointer is polymorphic, its dynamic type is the dynamic type of the target.

16.4.2.1 Pointer association status

A pointer may have a **pointer association status** of associated, disassociated, or undefined. Its association status may change during execution of a program. Unless a pointer is initialized (explicitly or by default), it has an initial association status of undefined. A pointer may be initialized to have an association status of disassociated.

NOTE 16.13

A pointer from a module program unit may be accessible in a subprogram via use association. Such pointers have a lifetime that is greater than targets that are declared in the subprogram, unless such targets are saved. Therefore, if such a pointer is associated with a local target, there is the possibility that when a procedure defined by the subprogram completes execution, the target will cease to exist, leaving the pointer "dangling". This standard considers such pointers to have an undefined association status. They are neither associated nor disassociated. They shall not be used again in the program until their status has been reestablished. There is no requirement on a processor to be able to detect when a pointer target ceases to exist.

16.4.2.1.1 Events that cause pointers to become associated

A pointer becomes associated when

(1) The pointer is allocated (6.3.1) as the result of the successful execution of an ALLOCATE statement referencing the pointer, or

(2) The pointer is pointer-assigned to a target (7.4.2) that is associated or is specified with the TARGET attribute and, if allocatable, is allocated.

16.4.2.1.2 Events that cause pointers to become disassociated

A pointer becomes disassociated when

(1) The pointer is nullified (6.3.2),

(2) The pointer is deallocated (6.3.3),

(3) The pointer is pointer-assigned (7.4.2) to a disassociated pointer, or

(4) The pointer is an ultimate component of an object of a type for which default initialization is specified for the component and

  (a) a procedure is invoked with this object as an actual argument corresponding to a nonpointer nonallocatable dummy argument with INTENT (OUT),

  (b) a procedure with this object as an unsaved nonpointer nonallocatable local object that is not accessed by use or host association is invoked, or

  (c) this object is allocated.

16.4.2.1.3 Events that cause the association status of pointers to become undefined

The association status of a pointer becomes undefined when

(1) The pointer is pointer-assigned to a target that has an undefined association status,

(2) The target of the pointer is deallocated other than through the pointer,

(3) The allocation transfer procedure (13.7.82) is executed with the pointer associated with the argument FROM and an object without the target attribute associated with the argument TO.

(4) Execution of a RETURN or END statement causes the pointer's target to become undefined (item (3) of 16.5.6),

(5) A procedure is terminated by execution of a RETURN or END statement and the pointer is declared or accessed in the subprogram that defines the procedure unless the pointer

  (a) Has the SAVE attribute,

  (b) Is in blank common,

  (c) Is in a named common block that appears in at least one other scoping unit that is in execution,

  (d) Is in the scoping unit of a module if the module also is accessed by another scoping unit that is in execution,

  (e) Is accessed by host association, or

  (f) Is the return value of a function declared to have the POINTER attribute,

(6) The pointer is an ultimate component of an object, default initialization is not specified for the component, and a procedure is invoked with this object as an actual argument corresponding to a dummy argument with INTENT(OUT), or

(7) A procedure is invoked with the pointer as an actual argument corresponding to a pointer dummy argument with INTENT(OUT).

16.4.2.1.4 Other events that change the association status of pointers

When a pointer becomes associated with another pointer by argument association, construct association, or host association, the effects on its association status are specified in 16.4.5.

While two pointers are name associated, storage associated, or inheritance associated, if the association status of one pointer changes, the association status of the other changes accordingly.

16.4.2.2 Pointer definition status

The definition status of a pointer is that of its target. If a pointer is associated with a definable target, the definition status of the pointer may be defined or undefined according to the rules for a variable (16.5).

16.4.2.3 Relationship between association status and definition status

If the association status of a pointer is disassociated or undefined, the pointer shall not be referenced or deallocated. Whatever its association status, a pointer always may be nullified, allocated, or pointer assigned. A nullified pointer is disassociated. When a pointer is allocated, it becomes associated but undefined. When a pointer is pointer assigned, its association and definition status become those of the specified *data-target* or *proc-target*.

16.4.3 Storage association

Storage sequences are used to describe relationships that exist among variables, common blocks, and result variables. **Storage association** is the association of two or more data objects that occurs when two or more storage sequences share or are aligned with one or more storage units.

16.4.3.1 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, a numeric storage unit, a file storage unit(9.2.4), or an unspecified storage unit. The sizes of the numeric storage unit, the character storage unit and the file storage unit are the value of constants in the ISO_FORTRAN_ENV intrinsic module (13.8.2).

In a storage association context

(1) A nonpointer scalar object of type default integer, default real, or default logical occupies a single **numeric storage unit**;

(2) A nonpointer scalar object of type double precision real or default complex occupies two contiguous numeric storage units;

(3) A nonpointer scalar object of type default character and character length *len* occupies *len* contiguous **character storage unit**s;

(4) A nonpointer scalar object of type character with the C character kind (15.1.1) and character length *len* occupies *len* contiguous **unspecified storage unit**s.

(5) A nonpointer scalar object of sequence type with no type parameters occupies a sequence of storage sequences corresponding to the sequence of its ultimate components;

(6) A nonpointer scalar object of any type not specified in items (1)-(5) occupies a single unspecified storage unit that is different for each case and each set of type parameter values, and that is different from the unspecified storage units of item (4);

(7) A nonpointer array occupies a sequence of contiguous storage sequences, one for each array element, in array element order (6.2.2.2); and

(8) A pointer occupies a single unspecified storage unit that is different from that of any non-pointer object and is different for each combination of type, type parameters, and rank.

A sequence of storage sequences forms a storage sequence. The order of the storage units in such a composite storage sequence is that of the individual storage units in each of the constituent storage sequences taken in succession, ignoring any zero-sized constituent sequences.

Each common block has a storage sequence (5.5.2.1).

16.4.3.2 Association of storage sequences

Two nonzero-sized storage sequences s_1 and s_2 are **storage 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.

Storage association also is defined between two zero-sized storage sequences, and between a zero-sized storage sequence and a storage unit. A zero-sized storage sequence in a sequence of storage sequences is storage associated with its successor, if any. If the successor is another zero-sized storage sequence, the two sequences are storage associated. If the successor is a nonzero-sized storage sequence, the zero-sized sequence is storage associated with the first storage unit of the successor. Two storage units that are each storage associated with the same zero-sized storage sequence are the same storage unit.

NOTE 16.14

Zero-sized objects may occur in a storage association context as the result of changing a parameter. For example, a program might contain the following declarations:

```
INTEGER, PARAMETER :: PROBSIZE = 10
INTEGER, PARAMETER :: ARRAYSIZE = PROBSIZE * 100
```

NOTE 16.14 (cont.)

```
REAL, DIMENSION (ARRAYSIZE) :: X
INTEGER, DIMENSION (ARRAYSIZE) :: IX
   ...
COMMON / EXAMPLE / A, B, C, X, Y, Z
EQUIVALENCE (X, IX)
   ...
```

If the first statement is subsequently changed to assign zero to PROBSIZE, the program still will conform to the standard.

16.4.3.3 Association of scalar data objects

Two scalar data objects are storage associated if their storage sequences are storage associated. Two scalar entities are **totally associated** if they have the same storage sequence. Two scalar entities are **partially associated** if they are associated without being totally associated.

The definition status and value of a data object affects the definition status and value of any storage associated entity. An EQUIVALENCE statement, a COMMON statement, or an ENTRY statement may cause storage association of storage sequences.

An EQUIVALENCE statement causes storage association of data objects only within one scoping unit, unless one of the equivalenced entities is also in a common block (5.5.1.1 and 5.5.2.1).

COMMON statements cause data objects in one scoping unit to become storage associated with data objects in another scoping unit.

A common block is permitted to contain a sequence of differing storage units. All scoping units that access named common blocks with the same name shall specify an identical sequence of storage units. Blank common blocks may be declared with differing sizes in different scoping units. For any two blank common blocks, the initial sequence of storage units of the longer blank common block shall be identical to the sequence of storage units of the shorter common block. If two blank common blocks are the same length, they shall have the same sequence of storage units.

An ENTRY statement in a function subprogram causes storage association of the result variables.

Partial association may exist only between

(1) An object of default character or character sequence type and an object of default character or character sequence type or

(2) An object of default complex, double precision real, or numeric sequence type and an object of default integer, default real, default logical, double precision real, default complex, or numeric sequence type.

For noncharacter entities, partial association may occur only through the use of COMMON, EQUIVALENCE, or ENTRY statements. For character entities, partial association may occur only through argument association or the use of COMMON or EQUIVALENCE statements.

NOTE 16.15

In the example:

```
REAL A (4), B
COMPLEX C (2)
```

NOTE 16.15 (cont.)

```
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. Although C (1) and C (2) are each storage associated with D, C (1) and C (2) are not storage 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.

NOTE 16.16

In the example:

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

A, B, and C are partially associated.

A storage unit shall not be explicitly initialized more than once in a program. Explicit initialization overrides default initialization, and default initialization for an object of derived type overrides default initialization for a component of the object (4.5.1). Default initialization may be specified for a storage unit that is storage associated provided the objects supplying the default initialization are of the same type and type parameters, and supply the same value for the storage unit.

16.4.4 Inheritance association

Inheritance association occurs between components of the parent component and components inherited by type extension into an extended type (4.5.6.1). This association is persistent; it is not affected by the accessibility of the inherited components.

16.4.5 Establishing associations

When an association is established between two entities by argument association, host association, or construct association, certain characteristics of the **associating entity** become those of the **pre-existing** entity.

For argument association, the associating entity is the dummy argument and the pre-existing entity is the actual argument. For host association, the associating entity is the entity in the host scoping unit and the pre-existing entity is the entity in the contained scoping unit. If the host scoping unit is a recursive procedure, the pre-existing entity that participates in the association is the one from the innermost procedure instance that invoked, directly or indirectly, the contained procedure. For construct association, the associating entity is identified by the associate name and the pre-existing entity is the selector.

When an association is established by argument association, host association, or construct association, the following applies:

(1) If the associating entity has the POINTER attribute, its pointer association status becomes the same as that of the pre-existing entity. If the pre-existing entity has a pointer association status of associated, the associating entity becomes pointer associated with the same target and, if they are arrays, the bounds of the associating entity become the same as those of the pre-existing entity.

(2) If the associating entity has the ALLOCATABLE attribute, its allocation status becomes the same as that of the pre-existing entity. If the pre-existing entity is allocated, the bounds (if it is an array), values of deferred type parameters, definition status, and value (if it is defined) become the same as those of the pre-existing entity. If the associating entity is polymorphic and the pre-existing entity is allocated, the dynamic type of the associating entity becomes the same as that of the pre-existing entity.

If the associating entity is neither a pointer nor allocatable, its definition status and value (if it is defined) become the same as those of the pre-existing entity. If the entities are arrays and the association is not argument association, the bounds of the associating entity become the same as those of the pre-existing entity.

16.5 Definition and undefinition of variables

A variable may be defined or may be undefined and its definition status may change during execution of a program. An action that causes a variable to become undefined does not imply that the variable was previously defined. An action that causes a variable to become defined does not imply that the variable was previously undefined.

16.5.1 Definition of objects and subobjects

Arrays, including sections, and variables of derived, character, or complex type are objects that consist of zero or more subobjects. Associations may be established between variables and subobjects and between subobjects of different variables. These subobjects may become defined or undefined.

(1) An array is defined if and only if all of its elements are defined.

(2) A derived-type scalar object is defined if and only if all of its nonpointer components are defined.

(3) A complex or character scalar object is defined if and only if all of its subobjects are defined.

(4) If an object is undefined, at least one (but not necessarily all) of its subobjects are undefined.

16.5.2 Variables that are always defined

Zero-sized arrays and zero-length strings are always defined.

16.5.3 Variables that are initially defined

The following variables are initially defined:

(1) Variables specified to have initial values by DATA statements,

(2) Variables specified to have initial values by type declaration statements,

(3) Nonpointer default-initialized subcomponents of variables that do not have the ALLOCATABLE or POINTER attribute, and are either saved or are declared in a main program, MODULE, or BLOCK DATA scoping unit,

(4) Variables that are always defined, and

(5) Variables with the BIND attribute that are initialized by means other than Fortran.

NOTE 16.17

Fortran code:

```
module mod
  integer, bind(c,name="blivet") :: foo
end module mod
```

C code:

```
int blivet = 123;
```

In the above example, the Fortran variable foo is initially defined to have the value 123 by means other than Fortran.

16.5.4 Variables that are initially undefined

All other variables are initially undefined.

16.5.5 Events that cause variables to become defined

Variables become defined as follows:

(1) Execution of an intrinsic assignment statement other than a masked array assignment or FORALL assignment statement causes the variable that precedes the equals to become defined. Execution of a defined assignment statement may cause all or part of the variable that precedes the equals to become defined.

(2) Execution of a masked array assignment or FORALL assignment statement may cause some or all of the array elements in the assignment statement to become defined (7.4.3).

eac0(mah4(2of)-1((t3)-339Asf)e339ttion)-1a04(inf)-i005B59a963niabav oft59(m)-1ueGBTf963nmause some

(9) Execution of a wait operation corresponding to an asynchronous input statement containing a SIZE= specifier causes the specified integer variable to become defined.

(10) 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.

(11) If an error, end-of-file, or end-of-record condition occurs ond error, end-of-file, o71(-)-1(fi)1(le,)-262-262(or)-245(

the real variable becomes defined and the real variable does not become undefined when the complex variable becomes defined. When a variable of type default complex is partially associated with another variable of type default complex, definition of one does not cause the other to become undefined.

(2) If the evaluation of a function may cause a variable 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, the variable becomes undefined when the expression is evaluated.

(3) When execution of an instance of a subprogram completes,

(a) its unsaved local variables become undefined,

(b) unsaved variables in a named common block that appears in the subprogram become undefined if they have been defined or redefined, unless another active scoping unit is referencing the common block,

(c) unsaved nonfinalizable local variables of a module become undefined unless another active scoping unit is referencing the module, and

NOTE 16.18

A module subprogram inherently references the module that is its host. Therefore, for processors that keep track of when modules are in use, a module is in use whenever any procedure in the module is active, even if no other active scoping units reference the module; this situation can arise if a module procedure is invoked via a procedure pointer, a type-bound procedure, or a companion processor.

(d) unsaved finalizable local variables of a module may be finalized if no other active scoping unit is referencing the module – following which they become undefined.

(4) 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 or namelist group of the statement become undefined.

(5) When an error condition, end-of-file condition, or end-of-record condition occurs during execution of an input/output statement and the statement contains any *io-implied-do*s, all of the *do-variable*s in the statement become undefined (9.10).

(6) Execution of a direct access input statement that specifies a record that has not been written previously causes all of the variables specified by the input list of the statement to become undefined.

(7) Execution of an INQUIRE statement may cause the NAME=, RECL=, and NEXTREC= variables to become undefined (9.9).

(8) When a character storage unit becomes undefined, all associated character storage units become undefined.

When a numeric storage unit becomes undefined, all associated numeric storage units become undefined unless the undefinition is a result of defining an associated numeric storage unit of different type (see (1) above).

When an entity of double precision real type becomes undefined, all totally associated entities of double precision real type become undefined.

When an unspecified storage unit becomes undefined, all associated unspecified storage units become undefined.

(9) When an allocatable entity is deallocated, it becomes undefined.

(10) When the allocation transfer procedure (13.5.16) causes the allocation status of an allocatable entity to become unallocated, the entity becomes undefined.

(11) Successful execution of an ALLOCATE statement for a nonzero-sized object that has a subcomponent for which default initialization has not been specified causes the subcomponent to become undefined.

(12) Execution of an INQUIRE statement causes all inquiry specifier variables to become undefined if an error condition exists, except for any variables in an IOSTAT= or IOMSG= specifier.

(13) When a procedure is invoked

(a) An optional dummy argument that is not associated with an actual argument becomes undefined;

(b) A dummy argument with INTENT (OUT) becomes undefined except for any nonpointer default-initialized subcomponents of the argument;

(c) An actual argument associated with a dummy argument with INTENT (OUT) becomes undefined except for any nonpointer default-initialized subcomponents of the argument;

(d) A subobject of a dummy argument that does not have INTENT (OUT) becomes undefined if the corresponding subobject of the actual argument is undefined; and

(e) The result variable of a function becomes undefined except for any of its nonpointer default-initialized subcomponents.

(14) When the association status of a pointer becomes undefined or disassociated (16.4.2.1.2-16.4.2.1.3), the pointer becomes undefined.

(15) When the execution of a FORALL construct completes, the *index-name*s become undefined.

(16) Execution of an asynchronous READ statement causes all of the variables specified by the input list or SIZE= specifier to become undefined. Execution of an asynchronous namelist READ statement causes any variable in the namelist group to become undefined if that variable will subsequently be defined during the execution of the READ statement or the corresponding WAIT operation.

(17) When execution of a RETURN or END statement causes a variable to become undefined, any variable of type C_PTR becomes undefined if its value is the C address of any part of the variable that becomes undefined.

(18) When a variable with the TARGET attribute is deallocated, any variable of type C_PTR becomes undefined if its value is the C address of any part of the variable that is deallocated.

NOTE 16.19

Execution of a defined assignment statement may leave all or part of the variable that precedes the equals undefined.

16.5.7 Variable definition context

Some variables are prohibited from appearing in a syntactic context that would imply definition or undefinition of the variable (5.1.2.7, 5.1.2.12, 12.6). The following are the contexts in which the appearance of a variable implies such definition or undefinition of the variable:

(1) The *variable* of an *assignment-stmt*,

(2) A *pointer-object* in a *pointer-assignment-stmt* or *nullify-stmt*,

(3) A *do-variable* in a *do-stmt* or *io-implied-do*,

(4) An *input-item* in a *read-stmt*,

(5) A *variable-name* in a *namelist-stmt* if the *namelist-group-name* appears in a NML= specifier in a *read-stmt*,

(6) An *internal-file-variable* in a *write-stmt*,

(7) An IOSTAT=, SIZE=, or IOMSG= specifier in an input/output statement,

(8) A definable variable in an INQUIRE statement,

(9) A *stat-variable*, *allocate-object*, or *errmsg10mEF327.9fiT,*

An opti9l argu94umentlir

dummy argument has the INTENT(OUT) or INTENT(INOUT) attribute, or

(11) A *variable* that is the *selector* in a SELECT TYPE or ASSOCIATE construct if the associate name of that construct appears in a variable definition context.

Annex A

(Informative)

Glossary of technical terms

The following is a list of the principal technical terms used in the standard and their definitions. A reference in parentheses immediately after a term is to the section where the term is defined or explained. The wording of a definition here is not necessarily the same as in the standard.

abstract type (4.5.6) : A type that has the ABSTRACT attribute. A nonpolymorphic object shall not be declared to be of abstract type. A polymorphic object shall not be constructed or allocated to have a dynamic abstract type.

action statement (2.1) : A single statement specifying or controlling a computational action (R214).

actual argument (12, 12.4.1) : An expression, a variable, a procedure, or an alternate return specifier that is specified in a procedure reference.

allocatable variable (5.1.2.2) : A variable having the ALLOCATABLE attribute. It may be referenced or defined only when it is allocated. If it is an array, it has a shape only when it is allocated. It may be a named variable or a structure component.

argument (12) : An actual argument or a dummy argument.

argument association (16.4.1.1) : The relationship between an actual argument and a dummy argument during the execution of a procedure reference.

array (2.4.5) : A set of scalar data, all of the same type and type parameters, whose individual elements are arranged in a rectangular pattern. It may be a named array, an array section, a structure component, a function value, or an expression. Its rank is at least one. Note that in FORTRAN 77, arrays were always named and never constants.

array element (2.4.5, 6.2.2) : One of the scalar data that make up an array that is either named or is a structure component.

array pointer (5.1.2.5.3) : A pointer to an array.

array section (2.4.5, 6.2.2.3) : A subobject that is an array and is not a structure component.

assignment statement (7.4.1.1) : A statement of the form "variable = expression".

associate name (8.1.4.1) : The name by which a selector of a SELECT TYPE or ASSOCIATE construct is known within the construct.

association (16.4) : Name association, pointer association, storage association, or inheritance association.

assumed-shape array (5.1.2.5.2) : A nonpointer dummy array that takes its shape from the associated actual argument.

assumed-size array (5.1.2.5.4) : A dummy array whose size is assumed from the associated actual argument. Its last upper bound is specified by an asterisk.

attribute (5) : A property of a data object that may be specified in a type declaration statement (R501).

automatic data object (5.1) : A data object that is a local entity of a subprogram, that is not a dummy argument, and that has a length type parameter or array bound that is specified by an expression that is not an initialization expression.

base type (4.5.6) : An extensible type that is not an extension of another type.

belong (8.1.6.4.3, 8.1.6.4.4) : If an EXIT or a CYCLE statement contains a construct name, the statement **belongs** to the DO construct using that name. Otherwise, it **belongs** to the innermost DO construct in which it appears.

binding label (15.4.1, 15.3.1) : A value of type default character that uniquely identifies how a variable, common block, subroutine, or function is known to a companion processor.

block (8.1) : A sequence of executable constructs embedded in another executable construct, bounded by statements that are particular to the construct, and treated as an integral unit.

block data program unit (11.3) : A program unit that provides initial values for data objects in named common blocks.

bounds (5.1.2.5.1) : For a named array, the limits within which the values of the subscripts of its array elements shall lie.

character (3.1) : A letter, digit, or other symbol.

class (5.1.1.2) : A class named N is the set of types extended from the type named N.

characteristics (12.2) :

(1) Of a procedure, its classification as a function or subroutine, whether it is pure, whether it is elemental, whether it has the BIND attribute, the value of its binding label, the characteristics of its dummy arguments, and the characteristics of its function result if it is a function.

(2) Of a dummy argument, whether it is a data object, is a procedure, is a procedure pointer, is an asterisk (alternate return indicator), or has the OPTIONAL attribute.

(3) Of a dummy data object, its type, type parameters, shape, the exact dependence of an array bound or type parameter on other entities, intent, whether it is optional, whether it is a pointer or a target, whether it is allocatable, whether it has the VALUE, ASYNCHRONOUS, or VOLATILE attributes, whether it is polymorphic, and whether the shape, size, or a type parameter is assumed.

(4) Of a dummy procedure or procedure pointer, whether the interface is explicit, the characteristics of the procedure if the interface is explicit, and whether it is optional.

(5) Of a function result, its type, type parameters, which type parameters are deferred, whether it is polymorphic, whether it is a pointer or allocatable, whether it is a procedure pointer, rank if it is a pointer or allocatable, shape if it is not a pointer or allocatable, the exact dependence of an array bound or type parameter on other entities, and whether the character length is assumed.

character length parameter (2.4.1.1) : The type parameter that specifies the number of characters for an entity of type character.

character string (4.4.4) : A sequence of characters numbered from left to right 1, 2, 3, ...

character storage unit (16.4.3.1) : The unit of storage for holding a scalar that is not a pointer and is of type default character and character length one.

collating sequence (4.4.4.3) : An ordering of all the different characters of a particular kind type parameter.

common block (5.5.2) : A block of physical storage that may be accessed by any of the scoping units in a program.

companion processor (2.5.10): A mechanism by which global data and procedures may be referenced or defined. It may be a mechanism that references and defines such entities by means other than Fortran. The procedures can be described by a C function prototype.

component (4.5) : A constituent of a derived type.

component order (4.5.3.5) : The ordering of the components of a derived type that is used for intrinsic formatted input/output and for structure constructors.

conformable (2.4.5) : Two arrays are said to be **conformable** if they have the same shape. A scalar is conformable with any array.

conformance (1.5) : A program conforms to the standard if it uses only those forms and relationships described therein and if the program has an interpretation according to the standard. A program unit conforms to the standard if it can be included in a program in a manner that allows the program to be standard conforming. A processor conforms to the standard if it executes standard-conforming programs in a manner that fulfills the interpretations prescribed in the standard and contains the capability of detection and reporting as listed in 1.5.

connected (9.4.3) :

(1) For an external unit, the property of referring to an external file.

(2) For an external file, the property of having an external unit that refers to it.

constant (2.4.3.1.2) : A data object whose value shall not change during execution of a program. It may be a named constant or a literal constant.

construct (7.4.3, 7.4.4, 8.1) : A sequence of statements starting with an ASSOCIATE, DO, FORALL, IF, SELECT CASE, SELECT TYPE, or WHERE statement and ending with the corresponding terminal statement.

construct entity (16) : An entity defined by a lexical token whose scope is a construct.

control mask (7.4.3) : In a WHERE statement or construct, an array of type logical whose value determines which elements of an array, in a *where-assignment-stmt*, will be defined.

data : Plural of datum.

data entity (2.4.3) : A data object, the result of the evaluation of an expression, or the result of the execution of a function reference (called the function result). A data entity has a type (either intrinsic or derived) and has, or may have, a data value (the exception is an undefined variable). Every data entity has a rank and is thus either a scalar or an array.

data object (2.4.3.1) : A data entity that is a constant, a variable, or a subobject of a constant.

data type (4) : See type.

datum : A single quantity that may have any of the set of values specified for its type.

decimal symbol (9.9.1.6, 10.5, 10.7.8) : The character that separates the whole and fractional parts in the decimal representation of a real number in a file. By default the decimal symbol is a decimal point (also known as a period). The current decimal symbol is determined by the current decimal edit mode.

declared type (5.1.1.2, 7.1.4) : The type that a data entity is declared to have. May differ from the type during execution (the dynamic type) for polymorphic data entities.

default initialization (4.5) : If initialization is specified in a type definition, an object of the type will be automatically initialized. Nonpointer components may be initialized with values by default; pointer components may be initially disassociated by default. Default initialization is not provided for objects of intrinsic type.

default-initialized (4.5.3.4) : A subcomponent is said to be default-initialized if it will be initialized by default initialization.

deferred binding (4.5.4) : A binding with the DEFERRED attribute. A deferred binding shall appear only in an abstract type definition (4.5.6).

deferred type parameter (4.3) : A length type parameter whose value is not specified in the declaration of an object, but instead is specified when the object is allocated or pointer-assigned.

definable (2.5.5) : A variable is **definable** if its value may be changed by the appearance of its designator on the left of an assignment statement. An allocatable variable that has not been allocated is an example of a data object that is not definable. An example of a subobject that is not definable is C (I) when C is an array that is a constant and I is an integer variable.

defined (2.5.5) : For a data object, the property of having or being given a valid value.

defined assignment statement (7.4.1.4, 12.3.2.1.2) : An assignment statement that is not an intrinsic assignment statement; it is defined by a subroutine and a generic interface that specifies ASSIGNMENT (=).

defined operation (7.1.3, 12.3.2.1.1) : An operation that is not an intrinsic operation and is defined by a function that is associated with a generic identifier.

deleted feature (1.8) : A feature in a previous Fortran standard that is considered to have been redundant and largely unused. See B.1 for a list of features that are in a previous Fortran standard, but are not in this standard. A feature designated as an obsolescent feature in the standard may become a deleted feature in the next revision.

derived type (2.4.1.2, 4.5) : A type whose data have components, each of which is either of intrinsic type or of another derived type.

designator (2.5.1) : A name, followed by zero or more component selectors, array section selectors, array element selectors, and substring selectors.

disassociated (2.4.6) : A disassociated pointer is not associated with a target. A pointer is disassociated following execution of a NULLIFY statement, following pointer assignment with a disassociated pointer, by default initialization, or by explicit initialization. A data pointer may also be disassociated by execution of a DEALLOCATE statement.

dummy argument (12, 12.5.2.1, 12.5.2.2, 12.5.2.4, 12.5.4) : An entity by which an associated actual argument is accessed during execution of a procedure.

dummy array : A dummy argument that is an array.

dummy data object (12.2.1.1, 12.4.1.2) : A dummy argument that is a data object.

dummy pointer : A dummy argument that is a pointer.

dummy procedure (12.1.2.3) : A dummy argument that is specified or referenced as a procedure.

dynamic type (5.1.1.2, 7.1.4) : The type of a data entity during execution of a program. The dynamic type of a data entity that is not polymorphic is the same as its declared type.

effective item (9.5.2) : A scalar object resulting from expanding an input/output list according to the rules in 9.5.2.

elemental (2.4.5, 7.4.1.4, 12.7) : An adjective applied to an operation, procedure, or assignment statement that is applied independently to elements of an array or corresponding elements of a set of conformable arrays and scalars.

entity : The term used for any of the following: a program unit, procedure, abstract interface, operator, generic interface, common block, external unit, statement function, type, data entity, statement label, construct, or namelist group.

executable construct (2.1) : An action statement (R214) or an ASSOCIATE, CASE, DO, FORALL, IF, SELECT TYPE, or WHERE construct.

executable statement (2.3.1) : An instruction to perform or control one or more computational actions.

explicit initialization (5.1) : Explicit initialization may be specified for objects of intrinsic or derived type in type declaration statements or DATA statements. An object of a derived type that specifies default initialization shall not appear in a DATA statement.

explicit interface (12.3.1) : If a procedure has explicit interface at the point of a reference to it, the processor is able to verify that the characteristics of the reference and declaration are related as required by this standard. A procedure has explicit interface if it is an internal procedure, a module procedure, an intrinsic procedure, an external procedure that has an interface body, a procedure reference in its own scoping unit, or a dummy procedure that has an interface body.

explicit-shape array (5.1.2.5.1) : A named array that is declared with explicit bounds.

expression (2.4.3.2, 7.1) : A sequence of operands, operators, and parentheses (R722). It may be a variable, a constant, a function reference, or may represent a computation.

extended type (4.5.6) : An extensible type that is an extension of another type. A type that is declared with the EXTENDS attribute.

extensible type (4.5.6) : A type from which new types may be derived using the EXTENDS attribute. A nonsequence type that does not have the BIND attribute.

extension type (4.5.6) : A base type is an extension type of itself only. An extended type is an extension type of itself and of all types for which its parent type is an extension.

extent (2.4.5) : The size of one dimension of an array.

external file (9.2) : A sequence of records that exists in a medium external to the program.

external linkage : The characteristic describing that a C entity is global to the program; defined in clause 6.2.2 of the C standard.

external procedure (2.2.3.1) : A procedure that is defined by an external subprogram or by a means other than Fortran.

external subprogram (2.2) : A subprogram that is not in a main program, module, or another subprogram. Note that a module is not called a subprogram. Note that in FORTRAN 77, a block data program unit is called a subprogram.

external unit (9.4) : A mechanism that is used to refer to an external file. It is identified by a nonnegative integer.

file (9) : An internal file or an external file.

file storage unit (9.2.4) : The unit of storage for an unformatted or stream file.

final subroutine (4.5.5) : A subroutine that is called automatically by the processor during finalization.

finalizable (4.5.5) : A type that has final subroutines, or that has a finalizable component. An object of finalizable type.

finalization (4.5.5.1) : The process of calling user-defined final subroutines immediately before destroying an object.

function (2.2.3) : A procedure that is invoked in an expression and computes a value which is then used in evaluating the expression.

function result (12.5.2.1) : The data object that returns the value of a function.

function subprogram (12.5.2.1) : A sequence of statements beginning with a FUNCTION statement that is not in an interface block and ending with the corresponding END statement.

generic identifier (12.3.2.1) : A lexical token that appears in an INTERFACE statement and is associated with all the procedures in the interface block.

generic interface (4.5.1, 12.3.2.1) : An interface specified by a generic procedure binding or a generic interface block.

generic interface block (12.3.2.1) : An interface block with a generic specification.

global entity (16.1) : An entity with an identifier whose scope is a program.

host (2.2) : Host scoping unit.

host association (16.4.1.3) : The process by which a contained scoping unit accesses entities of its host.

host scoping unit (2.2) : A scoping unit that immediately surrounds another scoping unit.

implicit interface (12.3.1) : For a procedure referenced in a scoping unit, the property of not having an explicit interface. A statement function always has an implicit interface

inherit (4.5.6) : To acquire from a parent. Type parameters, components, or procedure bindings of an extended type that are automatically acquired from its parent type without explicit declaration in the extended type are said to be inherited.

inheritance association (4.5.6.1, 16.4.4) : The relationship between the inherited components and the parent component in an extended type.

inquiry function (13.1) : A function that is either intrinsic or is defined in an intrinsic module and whose result depends on properties of one or more of its arguments instead of their values.

instance of a subprogram (12.5.2.3) : The copy of a subprogram that is created when a procedure defined by the subprogram is invoked.

intent (5.1.2.7) : An attribute of a dummy data object that indicates whether it is used to transfer data into the procedure, out of the procedure, or both.

interface block (12.3.2.1) : A sequence of statements from an INTERFACE statement to the corresponding END INTERFACE statement.

interface body (12.3.2.1) : A sequence of statements in an interface block from a FUNCTION or SUBROUTINE statement to the corresponding END statement.

interface of a procedure (12.3) : See procedure interface.

internal file (9.3) : A character variable that is used to transfer and convert data from internal storage to internal storage.

internal procedure (2.2.3.3) : A procedure that is defined by an internal subprogram.

internal subprogram (2.2) : A subprogram in a main program or another subprogram.

interoperable (15.2) : The property of a Fortran entity that ensures that an equivalent entity may be defined by means of C.

intrinsic (2.5.7) : An adjective that may be applied to types, operators, assignment statements, procedures, and modules. Intrinsic types, operators, and assignment statements are defined in this standard and may be used in any scoping unit without further definition or specification. Intrinsic procedures are defined in this standard or provided by a processor, and may be used in a scoping unit without further definition or specification. Intrinsic modules are defined in this standard or provided by a processor, and may be accessed by use association; procedures and types defined in an intrinsic module are not themselves intrinsic.

Intrinsic procedures and modules that are not defined in this standard are called nonstandard intrinsic procedures and modules.

invoke (2.2.3) :

(1) To call a subroutine by a CALL statement or by a defined assignment statement.

(2) To call a function by a reference to it by name or operator during the evaluation of an expression.

keyword (2.5.2) : A word that is part of the syntax of a statement or a name that is used to identify an item in a list.

kind type parameter (2.4.1.1, 4.4.1, 4.4.2, 4.4.3, 4.4.4, 4.4.5) : A parameter whose values label the available kinds of an intrinsic type.

label : See statement label.

length of a character string (4.4.4) : The number of characters in the character string.

lexical token (3.2) : A sequence of one or more characters with a specified interpretation.

line (3.3) : A sequence of 0 to 132 characters, which may contain Fortran statements, a comment, or an INCLUDE line.

linked (12.5.3) : When a C function with external linkage has the same binding label as a Fortran procedure, they are said to be linked. It is also possible for two Fortran entities to be linked.

literal constant (2.4.3.1.2, 4.4) : A constant without a name. Note that in FORTRAN 77, this was called simply a constant.

local entity (16.2) : An entity identified by a lexical token whose scope is a scoping unit.

local variable (2.4.3.1.1) : A variable local to a particular scoping unit; not imported through use or host association, not a dummy argument, and not a variable in common.

main program (2.3.4, 11.1) : A Fortran main program or a replacement defined by means other than Fortran.

many-one array section (6.2.2.3.2) : An array section with a vector subscript having two or more elements with the same value.

module (2.2.4, 11.2) : A program unit that contains or accesses definitions to be accessed by other program units.

module procedure (2.2.3.2) : A procedure that is defined by a module subprogram.

module subprogram (2.2) : A subprogram that is in a module but is not an internal subprogram.

name (3.2.1) : A lexical token consisting of a letter followed by up to 62 alphanumeric characters (letters, digits, and underscores). Note that in FORTRAN 77, this was called a symbolic name.

name association (16.4.1) : Argument association, use association, or host association.

named : Having a name. That is, in a phrase such as "named variable," the word "named" signifies that the variable name is not qualified by a subscript list, substring specification, and so on. For example, if X is an array variable, the reference "X" is a named variable while the reference "X(1)" is an object designator.

named constant (2.4.3.1.2) : A constant that has a name. Note that in FORTRAN 77, this was called a symbolic constant.

NaN (14.7) : A Not-a-Number value of IEEE arithmetic. It represents an undefined value or a value created by an invalid operation.

nonexecutable statement (2.3.1) : A statement used to configure the program environment in which computational actions take place.

numeric storage unit (16.4.3.1) : The unit of storage for holding a scalar that is not a pointer and is of type default real, default integer, or default logical.

numeric type (4.4) : Integer, real or complex type.

object (2.4.3.1) : Data object.

object designator (2.5.1) : A designator for a data object.

obsolescent feature (1.8) : A feature that is considered to have been redundant but that is still in frequent use.

operand (2.5.8) : An expression that precedes or succeeds an operator.

operation (7.1.2) : A computation involving one or two operands.

operator (2.5.8) : A lexical token that specifies an operation.

override (4.5.1, 4.5.6) : When explicit initialization or default initialization overrides default initialization, it is as if only the overriding initialization were specified. If a procedure is bound to an extensible type, it overrides the one that would have been inherited from the parent type.

parent component (4.5.6.1) : The component of an entity of extended type that corresponds to its inherited portion.

parent type (4.5.6) : The extensible type from which an extended type is derived.

passed-object dummy argument (4.5.1) : The dummy argument of a type-bound procedure or procedure pointer component that becomes associated with the object through which the procedure was invoked.

pointer (2.4.6) : An entity that has the POINTER attribute.

pointer assignment (7.4.2) : The pointer association of a pointer with a target by the execution of a pointer assignment statement or the execution of an assignment statement for a data object of derived type having the pointer as a subobject.

pointer assignment statement (7.4.2) : A statement of the form "pointer-object => target".

pointer associated (6.3, 7.4.2) : The relationship between a pointer and a target following a pointer assignment or a valid execution of an ALLOCATE statement.

pointer association (16.4.2) : The process by which a pointer becomes pointer associated with a target.

polymorphic (5.1.1.2) : Able to be of differing types during program execution. An object declared with the CLASS keyword is polymorphic.

preconnected (9.4.4) : A property describing a unit that is connected to an external file at the beginning of execution of a program. Such a unit may be specified in input/output statements without an OPEN statement being executed for that unit.

procedure (2.2.3, 12.1) : A computation that may be invoked during program execution. It may be a function or a subroutine. It may be an intrinsic procedure, an external procedure, a module procedure, an internal procedure, a dummy procedure, or a statement function. A subprogram may define more than one procedure if it contains ENTRY statements.

procedure designator (2.5.1) : A designator for a procedure.

procedure interface (12.3) : The characteristics of a procedure, the name of the procedure, the name of each dummy argument, and the generic identifiers (if any) by which it may be referenced.

processor (1.2) : The combination of a computing system and the mechanism by which programs are transformed for use on that computing system.

processor dependent (1.5) : The designation given to a facility that is not completely specified by this standard. Such a facility shall be provided by a processor, with methods or semantics determined by the processor.

program (2.2.1) : A set of program units that includes exactly one main program.

program unit (2.2) : The fundamental component of a program. A sequence of statements, comments, and INCLUDE lines. It may be a main program, a module, an external subprogram, or a block data program unit.

prototype : The C analog of a function interface body; defined in 6.7.5.3 of the C standard.

pure procedure (12.6) : A procedure that is a pure intrinsic procedure (13.1), is defined by a pure subprogram, or is a statement function that references only pure functions.

rank (2.4.4, 2.4.5) : The number of dimensions of an array. Zero for a scalar.

record (9.1) : A sequence of values or characters that is treated as a whole within a file.

reference (2.5.6) : The appearance of an object designator in a context requiring the value at that point during execution, the appearance of a procedure name, its operator symbol, or a defined assignment statement in a context requiring execution of the procedure at that point, or the appearance of a module name in a USE statement. Neither the act of defining a variable nor the appearance of the name of a procedure as an actual argument is regarded as a reference.

result variable (2.2.3, 12.5.2.1) : The variable that returns the value of a function.

rounding mode (14.3, 10.6.1.2.6) : The method used to choose the result of an operation that cannot be represented exactly. In IEEE arithmetic, there are four modes; nearest, towards zero, up (towards ∞), and down (towards $-\infty$). In addition, for input/output the two additional modes COMPATIBLE and PROCESSOR_DEFINED are provided.

scalar (2.4.4) :

(1) A single datum that is not an array.

(2) Not having the property of being an array.

scope (16) : That part of a program within which a lexical token has a single interpretation. It may be a program, a scoping unit, a construct, a single statement, or a part of a statement.

scoping unit (2.2) : One of the following:

(1) A program unit or subprogram, excluding any scoping units in it,

(2) A derived-type definition, or

(3) An interface body, excluding any scoping units in it.

section subscript (6.2.2) : A subscript, vector subscript, or subscript triplet in an array section selector.

selector (6.1.1, 6.1.2, 6.1.3, 8.1.3, 8.1.4) : A syntactic mechanism for designating

(1) Part of a data object. It may designate a substring, an array element, an array section, or a structure component.

(2) The set of values for which a CASE block is executed.

(3) The object whose type determines which branch of a SELECT TYPE construct is executed.

(4) The object that is associated with the *associate-name* in an ASSOCIATE construct.

shape (2.4.5) : The rank and extents of an array. The shape may be represented by the rank-one array whose elements are the extents in each dimension.

size (2.4.5) : The total number of elements of an array.

specification expression (7.1.6) : An expression with limitations that make it suitable for use in specifications such as length type parameters or array bounds.

specification function (7.1.6) : A nonintrinsic function that may be used in a specification expression.

standard-conforming program (1.5) : A program that uses only those forms and relationships described in this standard, and that has an interpretation according to this standard.

statement (3.3) : A sequence of lexical tokens. It usually consists of a single line, but a statement may be continued from one line to another and the semicolon symbol may be used to separate statements within a line.

statement entity (16) : An entity identified by a lexical token whose scope is a single statement or part of a statement.

statement function (12.5.4) : A procedure specified by a single statement that is similar in form to an assignment statement.

statement label (3.2.4) : A lexical token consisting of up to five digits that precedes a statement and may be used to refer to the statement.

storage association (16.4.3) : The relationship between two storage sequences if a storage unit of one is the same as a storage unit of the other.

storage sequence (16.4.3.1) : A sequence of contiguous storage units.

storage unit (16.4.3.1) : A character storage unit, a numeric storage unit, a file storage unit, or an unspecified storage unit.

stride (6.2.2.3.1) : The increment specified in a subscript triplet.

struct : The C analog of a sequence derived type; defined in 6.2.5 of the C standard.

structure (2.4.1.2) : A scalar data object of derived type.

structure component (6.1.2) : A part of an object of derived type. It may be referenced by an object designator.

structure constructor (4.5.9) : A syntactic mechanism for constructing a value of derived type.

subcomponent (6.1.2) : A subcomponent of an object of derived type is a component of that object or of a subobject of that object.

subobject (2.4.3.1) : A portion of a data object that may be referenced or defined independently of other portions. It may be an array element, an array section, a structure component, a substring, or the real or imaginary part of a complex object.

subprogram (2.2) : A function subprogram or a subroutine subprogram. Note that in FORTRAN 77, a block data program unit was called a subprogram.

subroutine (2.2.3) : A procedure that is invoked by a CALL statement or by a defined assignment statement.

subroutine subprogram (12.5.2.2) : A sequence of statements beginning with a SUBROUTINE statement that is not in an interface block and ending with the corresponding END statement.

subscript (6.2.2) : One of the list of scalar integer expressions in an array element selector. Note that in FORTRAN 77, the whole list was called the subscript.

subscript triplet (6.2.2) : An item in the list of an array section selector that contains a colon and specifies a regular sequence of integer values.

substring (6.1.1) : A contiguous portion of a scalar character string. Note that an array section can include a substring selector; the result is called an array section and not a substring.

target (2.4.6, 5.1.2.14, 6.3.1.2) : A data entity that has the TARGET attribute, or an entity that is associated with a pointer.

transformational function (13.1) : A function that is either intrinsic or is defined in an intrinsic module and that is neither an elemental function nor an inquiry function.

type (2.4.1) : A named category of data that is characterized by a set of values, together with a way to denote these values and a collection of operations that interpret and manipulate the values. The set of data values depends on the values of the type parameters.

type-bound procedure (4.5.4) : A procedure binding in a type definition. The procedure may be referenced by the *binding-name* via any object of that dynamic type, as a defined operator, by defined assignment, or as part of the finalization process.

type compatible (5.1.1.2) : All entities are type compatible with other entities of the same type. Unlimited polymorphic entities are type compatible with all entities; other polymorphic entities are type compatible with entities whose dynamic type is an extension type of the polymorphic entity's declared type.

type declaration statement (5) : An INTEGER, REAL, DOUBLE PRECISION, COMPLEX, CHARACTER, LOGICAL, or TYPE (*type-name*) statement.

type parameter (2.4.1.1) : A parameter of a data type. KIND and LEN are the type parameters of intrinsic types. The type parameters of a derived type are defined in the derived-type definition.

type parameter order (4.5.2.1) : The ordering of the type parameters of a derived type that is used for derived-type specifiers.

ultimate component (4.5) : For a structure, a component that is of intrinsic type, has the ALLOCATABLE attribute, or has the POINTER attribute, or an ultimate component of a derived-type component that does not have the POINTER attribute or the ALLOCATABLE attribute.

undefined (2.5.5) : For a data object, the property of not having a determinate value.

unsigned : A qualifier of a C numeric type indicating that it is comprised only of nonnegative values; defined in 6.2.5 of the C standard. There is nothing analogous in Fortran.

unspecified storage unit (16.4.3.1) : A unit of storage for holding a pointer or a scalar that is not a pointer and is of type other than default integer, default character, default real, double precision real, default logical, or default complex.

use association (16.4.1.2) : The association of names in different scoping units specified by a USE statement.

variable (2.4.3.1.1) : A data object whose value can be defined and redefined during the execution of a program. It may be a named data object, an array element, an array section, a structure component, or a substring. Note that in FORTRAN 77, a variable was always scalar and named.

vector subscript (6.2.2.3.2) : A section subscript that is an integer expression of rank one.

void : A C type comprising an empty set of values; defined in 6.2.5 of the C standard. There is nothing analogous in Fortran.

whole array (6.2.1) : A named array.

Annex B

(Informative)

Decremental features

B.1 Deleted features

The deleted features are those features of Fortran 90 that were redundant and are considered largely unused. Section 1.8.1 describes the nature of the deleted features. The Fortran 90 features that are not contained in Fortran 95 or this standard are the following:

(1) Real and double precision DO variables.

The ability present in FORTRAN 77, and for consistency also in Fortran 90, for a DO variable to be of type real or double precision in addition to type integer, has been deleted. A similar result can be achieved by using a DO construct with no loop control and the appropriate exit test.

(2) Branching to an END IF statement from outside its block.

In FORTRAN 77, and for consistency also in Fortran 90, it was possible to branch to an END IF statement from outside the IF construct; this has been deleted. A similar result can be achieved by branching to a CONTINUE statement that is immediately after the END IF statement.

(3) PAUSE statement.

The PAUSE statement, present in FORTRAN 66, FORTRAN 77 and for consistency also in Fortran 90, has been deleted. A similar result can be achieved by writing a message to the appropriate unit, followed by reading from the appropriate unit.

(4) ASSIGN and assigned GO TO statements and assigned format specifiers.

The ASSIGN statement and the related assigned GO TO statement, present in FORTRAN 66, FORTRAN 77 and for consistency also in Fortran 90, have been deleted. Further, the ability to use an assigned integer as a format, present in FORTRAN 77 and Fortran 90, has been deleted. A similar result can be achieved by using other control constructs instead of the assigned GOTO statement and by using a default character variable to hold a format specification instead of using an assigned integer.

(5) H edit descriptor.

In FORTRAN 77, and for consistency also in Fortran 90, there was an alternative form of character string edit descriptor, which had been the only such form in FORTRAN 66; this has been deleted. A similar result can be achieved by using a character string edit descriptor.

The following is a list of the previous editions of the international Fortran standard, along with their informal names:

ISO/IEC 1539:1972	FORTRAN 66
ISO/IEC 1539:1978	FORTRAN 77
ISO/IEC 1539:1991	Fortran 90
ISO/IEC 1539-1:1997	Fortran 95

See the Fortran 90 standard for detailed rules of how these deleted features work.

B.2 Obsolescent features

The obsolescent features are those features of Fortran 90 that were redundant and for which better methods were available in Fortran 90. Section 1.8.2 describes the nature of the obsolescent features. The obsolescent features in this standard are the following:

(1) Arithmetic IF — use the IF statement (8.1.2.4) or IF construct (8.1.2).

(2) Shared DO termination and termination on a statement other than END DO or CONTINUE — use an END DO or a CONTINUE statement for each DO statement.

(3) Alternate return — see B.2.1.

(4) Computed GO TO statement — see B.2.2.

(5) Statement functions — see B.2.3.

(6) DATA statements amongst executable statements — see B.2.4.

(7) Assumed length character functions — see B.2.5.

(8) Fixed form source — see B.2.6.

(9) CHARACTER* form of CHARACTER declaration — see B.2.7.

B.2.1 Alternate return

An alternate return introduces labels into an argument list to allow the called procedure to direct the execution of the caller upon return. The same effect can be achieved with a return code that is used in a CASE construct on return. This avoids an irregularity in the syntax and semantics of argument association. For example,

```
CALL SUBR_NAME (X, Y, Z, *100, *200, *300)
```

may be replaced by

```
CALL SUBR_NAME (X, Y, Z, RETURN_CODE)
SELECT CASE (RETURN_CODE)
CASE (1)
...
CASE (2)
...
CASE (3)
...
CASE DEFAULT
...
END SELECT
```

B.2.2 Computed GO TO statement

The computed GO TO has been superseded by the CASE construct, which is a generalized, easier to use and more efficient means of expressing the same computation.

B.2.3 Statement functions

Statement functions are subject to a number of nonintuitive restrictions and are a potential source of error because their syntax is easily confused with that of an assignment statement.

The internal function is a more generalized form of the statement function and completely supersedes it.

B.2.4 DATA statements among executables

The statement ordering rules of FORTRAN 66, and hence of FORTRAN 77 and Fortran 90 for compatibility, allowed DATA statements to appear anywhere in a program unit after the specification statements. The ability to position DATA statements amongst executable statements is very rarely used, is unnecessary and is a potential source of error.

B.2.5 Assumed character length functions

Assumed character length for functions is an irregularity in the language in that elsewhere in Fortran the philosophy is that the attributes of a function result depend only on the actual arguments of the invocation and on any data accessible by the function through host or use association. Some uses of this facility can be replaced with an automatic character length function, where the length of the function result is declared in a specification expression. Other uses can be replaced by the use of a subroutine whose arguments correspond to the function result and the function arguments.

Note that dummy arguments of a function may be assumed character length.

B.2.6 Fixed form source

Fixed form source was designed when the principal machine-readable input medium for new programs was punched cards. Now that new and amended programs are generally entered via keyboards with screen displays, it is an unnecessary overhead, and is potentially error-prone, to have to locate positions 6, 7, or 72 on a line. Free form source was designed expressly for this more modern technology.

It is a simple matter for a software tool to convert from fixed to free form source.

B.2.7 CHARACTER* form of CHARACTER declaration

Fortran 90 had two different forms of specifying the length selector in CHARACTER declarations. The older form (CHARACTER*char-length) is redundant.

Annex C

(Informative)

Extended notes

C.1 Section 4 notes

C.1.1 Intrinsic and derived types (4.4, 4.5)

FORTRAN 77 provided only types explicitly defined in the standard (logical, integer, real, double precision, complex, and character). This standard provides those intrinsic types and provides derived types to allow the creation of new types. A derived-type definition specifies a data structure consisting of components of intrinsic types and of derived types. Such a type definition does not represent a data object, but rather, a template for declaring named objects of that derived type. For example, the definition

```
TYPE POINT
   INTEGER X_COORD
   INTEGER Y_COORD
END TYPE POINT
```

specifies a new derived type named POINT which is composed of two components of intrinsic type integer (X_COORD and Y_COORD). The statement TYPE (POINT) FIRST, LAST declares two data objects, FIRST and LAST, that can hold values of type POINT.

FORTRAN 77 provided REAL and DOUBLE PRECISION intrinsic types as approximations to mathematical real numbers. This standard generalizes REAL as an intrinsic type with a type parameter that selects the approximation method. The type parameter is named kind and has values that are processor dependent. DOUBLE PRECISION is treated as a synonym for REAL (k), where k is the implementation-defined kind type parameter value KIND (0.0D0).

Real literal constants may be specified with a kind type parameter to ensure that they have a particular kind type parameter value (4.4.2).

For example, with the specifications

```
INTEGER Q
PARAMETER (Q = 8)
REAL (Q) B
```

the literal constant 10.93_Q has the same precision as the variable B.

FORTRAN 77 did not allow zero-length character strings. They are permitted by this standard (4.4.4).

Objects are of different derived type if they are declared using different derived-type definitions. For example,

```
TYPE APPLES
```

```
   INTEGER NUMBER
END TYPE APPLES
TYPE ORANGES
   INTEGER NUMBER
END TYPE ORANGES
TYPE (APPLES) COUNT1
TYPE (ORANGES) COUNT2
COUNT1 = COUNT2 ! Erroneous statement mixing apples and oranges
```

Even though all components of objects of type APPLES and objects of type ORANGES have identical intrinsic types, the objects are of different types.

C.1.2 Selection of the approximation methods (4.4.2)

One can select the real approximation method for an entire program through the use of a module and the parameterized real type. This is accomplished by defining a named integer constant to have a particular kind type parameter value and using that named constant in all real, complex, and derived-type declarations. For example, the specification statements

```
INTEGER, PARAMETER :: LONG_FLOAT = 8
REAL (LONG_FLOAT) X, Y
COMPLEX (LONG_FLOAT) Z
```

specify that the approximation method corresponding to a kind type parameter value of 8 is supplied for the data objects X, Y, and Z in the program unit. The kind type parameter value LONG_FLOAT can be made available to an entire program by placing the INTEGER specification statement in a module and accessing the named constant LONG_FLOAT with a USE statement. Note that by changing 8 to 4 once in the module, a different approximation method is selected.

To avoid the use of the processor-dependent values 4 or 8, replace 8 by KIND (0.0) or KIND (0.0D0). Another way to avoid these processor-dependent values is to select the kind value using the intrinsic inquiry function SELECTED_REAL_KIND. This function, given integer arguments P and R specifying minimum requirements for decimal precision and decimal exponent range, respectively, returns the kind type parameter value of the approximation method that has at least P decimal digits of precision and at least a range for positive numbers of $10^{-\mathrm{R}}$ to 10^{R}. In the above specification statement, the 8 may be replaced by, for instance, SELECTED_REAL_KIND (10, 50), which requires an approximation method to be selected with at least 10 decimal digits of precision and a range from 10^{-50} to 10^{50}. There are no magnitude or ordering constraints placed on kind values, in order that implementers may have flexibility in assigning such values and may add new kinds without changing previously assigned kind values.

As kind values have no portable meaning, a good practice is to use them in programs only through named constants as described above (for example, SINGLE, IEEE_SINGLE, DOUBLE, and QUAD), rather than using the kind values directly.

C.1.3 Type extension and component accessibility (4.5.1.1, 4.5.3)

The default accessibility of an extended type may be specified in the type definition. The accessibility of its components may be specified individually.

```
  module types
```

```
  type base_type
    private                !-- Sets default accessibility
    integer :: i           !-- a private component
    integer, private :: j  !-- another private component
    integer, public :: k   !-- a public component
  end type base_type

  type, extends(base_type) :: my_type
    private               !-- Sets default for components declared in my_type
    integer :: l          !-- A private component.
    integer, public :: m  !-- A public component.
  end type my_type

end module types

subroutine sub
  use types
  type (my_type) :: x

  ....

  call another_sub( &
    x%base_type,    &  !-- ok because base_type is a public subobject of x
    x%base_type%k,  &  !-- ok because x%base_type is ok and has k as a
                       !-- public component.
```

```
      END SUBROUTINE RENDER_X
   END INTERFACE
```

We can declare a nonabstract type by extending the abstract type:

```
   TYPE, EXTENDS(DRAWABLE_OBJECT) :: DRAWABLE_TRIANGLE ! Not ABSTRACT
      REAL, DIMENSION(2,3) :: VERTICES ! In relation to centroid
   CONTAINS
      PROCEDURE, PASS(TRIANGLE) :: RENDER=>RENDER_TRIANGLE_X
   END TYPE DRAWABLE_TRIANGLE
```

The actual drawing procedure will draw a triangle in WINDOW with vertices at x coordinates `TRIANGLE%POSITION(1)+TRIANGLE%VERTICES(1,:)` and y coordinates `TRIANGLE%POSITION(2)+TRIANGLE%VERTICES(2,:)`:

```
   SUBROUTINE RENDER_TRIANGLE_X(TRIANGLE, WINDOW)
      CLASS(DRAWABLE_TRIANGLE), INTENT(IN) :: TRIANGLE
      CLASS(X_WINDOW), INTENT(INOUT) :: WINDOW
      ...
   END SUBROUTINE RENDER_TRIANGLE_X
```

C.1.5 Pointers (4.5.1)

Pointers are names that can change dynamically their association with a target object. In a sense, a normal variable is a name with a fixed association with a particular object. A normal variable name refers to the same storage space throughout the lifetime of the variable. A pointer name may refer to different storage space, or even no storage space, at different times. A variable may be considered to be a descriptor for space to hold values of the appropriate type, type parameters, and array rank such that the values stored in the descriptor are fixed when the variable is created. A pointer also may be considered to be a descriptor, but one whose values may be changed dynamically so as to describe different pieces of storage. When a pointer is declared, space to hold the descriptor is created, but the space for the target object is not created.

A derived type may have one or more components that are defined to be pointers. It may have a component that is a pointer to an object of the same derived type. This "recursive" data definition allows dynamic data structures such as linked lists, trees, and graphs to be constructed. For example:

```
TYPE NODE          ! Define a ''recursive'' type
   INTEGER :: VALUE = 0
   TYPE (NODE), POINTER :: NEXT_NODE => NULL ( )
END TYPE NODE

TYPE (NODE), TARGET :: HEAD       ! Automatically initialized
TYPE (NODE), POINTER :: CURRENT, TEMP  ! Declare pointers
INTEGER :: IOEM, K

CURRENT => HEAD                    ! CURRENT points to head of list
```

```
DO
   READ (*, *, IOSTAT = IOEM) K  ! Read next value, if any
   IF (IOEM /= 0) EXIT
   ALLOCATE (TEMP)               ! Create new cell each iteration
   TEMP % VALUE = K              ! Assign value to cell
   CURRENT % NEXT_NODE => TEMP   ! Attach new cell to list
   CURRENT => TEMP               ! CURRENT points to new end of list
END DO
```

A list is now constructed and the last linked cell contains a disassociated pointer. A loop can be used to "walk through" the list.

```
CURRENT => HEAD
DO
   IF (.NOT. ASSOCIATED (CURRENT % NEXT_NODE)) EXIT
   CURRENT => CURRENT % NEXT_NODE
   WRITE (*, *) CURRENT % VALUE
END DO
```

C.1.6 Structure constructors and generic names

A generic name may be the same as a type name. This can be used to emulate user-defined structure constructors for that type, even if the type has private components. For example:

```
MODULE mytype_module
  TYPE mytype
    PRIVATE
    COMPLEX value
    LOGICAL exact
  END TYPE
  INTERFACE mytype
    MODULE PROCEDURE int_to_mytype
  END INTERFACE
  ! Operator definitions etc.
  ...
CONTAINS
  TYPE(mytype) FUNCTION int_to_mytype(i)
    INTEGER,INTENT(IN) :: i
    int_to_mytype%value = i
    int_to_mytype%exact = .TRUE.
  END FUNCTION
  ! Procedures to support operators etc.
  ...
END
```

```
PROGRAM example
  USE mytype_module
  TYPE(mytype) x
  x = mytype(17)
END
```

The type name may still be used as a generic name if the type has type parameters. For example:

```
MODULE m
  TYPE t(kind)
    INTEGER, KIND :: kind
    COMPLEX(kind) value
  END TYPE
  INTEGER,PARAMETER :: single = KIND(0.0), double = KIND(0d0)
  INTERFACE t
    MODULE PROCEDURE real_to_t1, dble_to_t2, int_to_t1, int_to_t2
  END INTERFACE
  ...
CONTAINS
  TYPE(t(single)) FUNCTION real_to_t1(x)
    REAL(single) x
    real_to_t1%value = x
  END FUNCTION
  TYPE(t(double)) FUNCTION dble_to_t2(x)
    REAL(double) x
    dble_to_t2%value = x
  END FUNCTION
  TYPE(t(single)) FUNCTION int_to_t1(x,mold)
    INTEGER x
    TYPE(t(single)) mold
    int_to_t1%value = x
  END FUNCTION
  TYPE(t(double)) FUNCTION int_to_t2(x,mold)
    INTEGER x
    TYPE(t(double)) mold
    int_to_t2%value = x
  END FUNCTION
  ...
END

PROGRAM example
  USE m
  TYPE(t(single)) x
  TYPE(t(double)) y
```

```
  x = t(1.5)                 ! References real_to_t1
  x = t(17,mold=x)           ! References int_to_t1
  y = t(1.5d0)               ! References dble_to_t2
  y = t(42,mold=y)           ! References int_to_t2
  y = t(kind(0d0)) ((0,1))  ! Uses the structure constructor for type t
END
```

C.1.7 Generic type-bound procedures

Example of a derived type with generic type-bound procedures:

The only difference between this example and the same thing rewritten to use generic interface blocks is that with type-bound procedures,

```
    USE(rational_numbers),ONLY :: rational
```

does not block the type-bound procedures; the user still gets access to the defined assignment and extended operations.

```
MODULE rational_numbers
  IMPLICIT NONE
  PRIVATE
  TYPE,PUBLIC :: rational
    PRIVATE
    INTEGER n,d
  CONTAINS
    ! ordinary type-bound procedure
    PROCEDURE :: real => rat_to_real
    ! specific type-bound procedures for generic support
    PROCEDURE,PRIVATE :: rat_asgn_i, rat_plus_rat, rat_plus_i
    PROCEDURE,PRIVATE,PASS(b) :: i_plus_rat
    ! generic type-bound procedures
    GENERIC :: ASSIGNMENT(=) => rat_asgn_i
    GENERIC :: OPERATOR(+) => rat_plus_rat, rat_plus_i, i_plus_rat
  END TYPE
CONTAINS
  ELEMENTAL REAL FUNCTION rat_to_real(this) RESULT(r)
    CLASS(rational),INTENT(IN) :: this
    r = REAL(this%n)/this%d
  END FUNCTION
  ELEMENTAL SUBROUTINE rat_asgn_i(a,b)
    CLASS(rational),INTENT(OUT) :: a
    INTEGER,INTENT(IN) :: b
    a%n = b
    a%d = 1
  END SUBROUTINE
```

```
  ELEMENTAL TYPE(rational) FUNCTION rat_plus_i(a,b) RESULT(r)
    CLASS(rational),INTENT(IN) :: a
    INTEGER,INTENT(IN) :: b
    r%n = a%n + b*a%d
    r%d = a%d
  END FUNCTION
  ELEMENTAL TYPE(rational) FUNCTION i_plus_rat(a,b) RESULT(r)
    INTEGER,INTENT(IN) :: a
    CLASS(rational),INTENT(IN) :: b
    r%n = b%n + a*b%d
    r%d = b%d
  END FUNCTION
  ELEMENTAL TYPE(rational) FUNCTION rat_plus_rat(a,b) RESULT(r)
    CLASS(rational),INTENT(IN) :: a,b
    r%n = a%n*b%d + b%n*a%d
    r%d = a%d*b%d
  END FUNCTION
END
```

C.1.8 Final subroutines (4.5.5, 4.5.5.1, 4.5.5.2, 4.5.5.3)

Example of a parameterized derived type with final subroutines:

```
MODULE m
  TYPE t(k)
    INTEGER, KIND :: k
    REAL(k),POINTER :: vector(:) => NULL()
  CONTAINS
    FINAL :: finalize_t1s, finalize_t1v, finalize_t2e
  END TYPE
CONTAINS
  SUBROUTINE finalize_t1s(x)
    TYPE(t(KIND(0.0))) x
    IF (ASSOCIATED(x%vector)) DEALLOCATE(x%vector)
  END SUBROUTINE
  SUBROUTINE finalize_t1v(x)
    TYPE(t(KIND(0.0))) x(:)
    DO i=LBOUND(x,1),UBOUND(x,1)
      IF (ASSOCIATED(x(i)%vector)) DEALLOCATE(x(i)%vector)
    END DO
  END SUBROUTINE
  ELEMENTAL SUBROUTINE finalize_t2e(x)
    TYPE(t(KIND(0.0d0))),INTENT(INOUT) :: x
    IF (ASSOCIATED(x%vector)) DEALLOCATE(x%vector)
  END SUBROUTINE
```

```
END MODULE

SUBROUTINE example(n)
  USE m
  TYPE(t(KIND(0.0))) a,b(10),c(n,2)
  TYPE(t(KIND(0.0d0))) d(n,n)
  ...
  ! Returning from this subroutine will effectively do
  !    CALL finalize_t1s(a)
  !    CALL finalize_t1v(b)
  !    CALL finalize_t2e(d)
  ! No final subroutine will be called for variable C because the user
  ! omitted to define a suitable specific procedure for it.
END SUBROUTINE
```

Example of extended types with final subroutines:

```
MODULE m
  TYPE t1
    REAL a,b
  END TYPE
  TYPE,EXTENDS(t1) :: t2
    REAL,POINTER :: c(:),d(:)
  CONTAINS
    FINAL :: t2f
  END TYPE
  TYPE,EXTENDS(t2) :: t3
    REAL,POINTER :: e
  CONTAINS
    FINAL :: t3f
  END TYPE
  ...
CONTAINS
  SUBROUTINE t2f(x) ! Finalizer for TYPE(t2)'s extra components
    TYPE(t2) :: x
    IF (ASSOCIATED(x%c)) DEALLOCATE(x%c)
    IF (ASSOCIATED(x%d)) DEALLOCATE(x%d)
  END SUBROUTINE
  SUBROUTINE t3f(y) ! Finalizer for TYPE(t3)'s extra components
    TYPE(t3) :: y
    IF (ASSOCIATED(y%e)) DEALLOCATE(y%e)
  END SUBROUTINE
END MODULE

SUBROUTINE example
```

```
  USE m
  TYPE(t1) x1
  TYPE(t2) x2
  TYPE(t3) x3
  ...
  ! Returning from this subroutine will effectively do
  !    ! Nothing to x1; it is not finalizable
  !    CALL t2f(x2)
  !    CALL t3f(x3)
  !    CALL t2f(x3%t2)
END SUBROUTINE
```

C.2 Section 5 notes

C.2.1 The POINTER attribute (5.1.2.11)

The POINTER attribute shall be specified to declare a pointer. The type, type parameters, and rank, which may be specified in the same statement or with one or more attribute specification statements, determine the characteristics of the target objects that may be associated with the pointers declared in the statement. An obvious model for interpreting declarations of pointers is that such declarations create for each name a descriptor. Such a descriptor includes all the data necessary to describe fully and locate in memory an object and all subobjects of the type, type parameters, and rank specified. The descriptor is created empty; it does not contain values describing how to access an actual memory space. These descriptor values will be filled in when the pointer is associated with actual target space.

The following example illustrates the use of pointers in an iterative algorithm:

```
PROGRAM DYNAM_ITER
   REAL, DIMENSION (:, :), POINTER :: A, B, SWAP  ! Declare pointers
   ...
   READ (*, *) N, M
   ALLOCATE (A (N, M), B (N, M))  ! Allocate target arrays
   ! Read values into A
   ...
   ITER: DO
      ...
      ! Apply  transformation of values in A to produce values in B
      ...
      IF (CONVERGED) EXIT ITER
      ! Swap A and B
      SWAP => A; A => B; B => SWAP
   END DO ITER
   ...
END PROGRAM DYNAM_ITER
```

C.2.2 The TARGET attribute (5.1.2.14)

The TARGET attribute shall be specified for any nonpointer object that may, during the execution of the program, become associated with a pointer. This attribute is defined primarily for optimization purposes. It allows the processor to assume that any nonpointer object not explicitly declared as a target may be referred to only by way of its original declared name. It also means that implicitly-declared objects shall not be used as pointer targets. This will allow a processor to perform optimizations that otherwise would not be possible in the presence of certain pointers.

The following example illustrates the use of the TARGET attribute in an iterative algorithm:

```
PROGRAM ITER
   REAL, DIMENSION (1000, 1000), TARGET :: A, B
   REAL, DIMENSION (:, :), POINTER       :: IN, OUT, SWAP
   ...
   ! Read values into A
   ...
   IN => A            ! Associate IN with target A
   OUT => B           ! Associate OUT with target B
   ...
   ITER:DO
      ...
      ! Apply transformation of IN values to produce OUT
      ...
      IF (CONVERGED) EXIT ITER
      ! Swap IN and OUT
      SWAP => IN; IN => OUT; OUT => SWAP
   END DO ITER
   ...
END PROGRAM ITER
```

C.2.3 The VOLATILE attribute (5.1.2.16)

The following example shows the use of a variable with the VOLATILE attribute to communicate with an asynchronous process, in this case the operating system. The program detects a user keystroke on the terminal and reacts at a convenient point in its processing.

The VOLATILE attribute is necessary to prevent an optimizing compiler from storing the communication variable in a register or from doing flow analysis and deciding that the EXIT statement can never be executed.

```
SUBROUTINE TERMINATE_ITERATIONS

  LOGICAL, VOLATILE  ::   USER_HIT_ANY_KEY

!    Have the OS start to look for a user keystroke and set the variable
!    "USER_HIT_ANY_KEY" to TRUE as soon as it detects a keystroke.
!    This pseudo call is operating system dependent.
```

```
  CALL OS_BEGIN_DETECT_USER_KEYSTROKE( USER_HIT_ANY_KEY )

  USER_HIT_ANY_KEY = .FALSE.      ! This will ignore any recent keystrokes

  PRINT *, " Hit any key to terminate iterations!"

  DO I = 1,100
     ...                      ! Compute a value for R
     PRINT *, I, R
     IF (USER_HIT_ANY_KEY)      EXIT
  ENDDO

! Have the OS stop looking for user keystrokes
  CALL OS_STOP_DETECT_USER_KEYSTROKE

END SUBROUTINE TERMINATE_ITERATIONS
```

C.3 Section 6 notes

C.3.1 Structure components (6.1.2)

Components of a structure are referenced by writing the components of successive levels of the structure hierarchy until the desired component is described. For example,

```
TYPE ID_NUMBERS
   INTEGER SSN
   INTEGER EMPLOYEE_NUMBER
END TYPE ID_NUMBERS

TYPE PERSON_ID
   CHARACTER (LEN=30) LAST_NAME
   CHARACTER (LEN=1) MIDDLE_INITIAL
   CHARACTER (LEN=30) FIRST_NAME
   TYPE (ID_NUMBERS) NUMBER
END TYPE PERSON_ID

TYPE PERSON
   INTEGER AGE
   TYPE (PERSON_ID) ID
END TYPE PERSON

TYPE (PERSON) GEORGE, MARY

PRINT *, GEORGE % AGE             ! Print the AGE component
```

```
PRINT *, MARY % ID % LAST_NAME    ! Print LAST_NAME of MARY
PRINT *, MARY % ID % NUMBER % SSN ! Print SSN of MAR
PRINT *, GEORGE % ID % NUMBER  ! Print SSN and EMPLOYEE_NUMBER of GEORGE
```

A structure component may be a data object of intrinsic type as in the case of GEORGE % AGE or it may be of derived type as in the case of GEORGE % ID % NUMBER. The resultant component may be a scalar or an array of intrinsic or derived type.

```
TYPE LARGE
   INTEGER ELT (10)
   INTEGER VAL
END TYPE LARGE

TYPE (LARGE) A (5)       ! 5 element array, each of whose elements
                         ! includes a 10 element array ELT and
                         ! a scalar VAL.
PRINT *, A (1)           ! Prints 10 element array ELT and scalar VAL.
PRINT *, A (1) % ELT (3) ! Prints scalar element 3
                         ! of array element 1 of A.
PRINT *, A (2:4) % VAL   ! Prints scalar VAL for array elements
                         ! 2 to 4 of A.
```

Components of an object of extensible type that are inherited from the parent type may be accessed as a whole by using the parent component name, or individually, either with or without qualifying them by the parent component name.

For example:

```
  TYPE POINT            ! A base type
    REAL :: X, Y
  END TYPE POINT
  TYPE, EXTENDS(POINT) :: COLOR_POINT  ! An extension of TYPE(POINT)
    ! Components X and Y, and component name POINT, inherited from parent
    INTEGER :: COLOR
  END TYPE COLOR_POINT

  TYPE(POINT) :: PV = POINT(1.0, 2.0)
  TYPE(COLOR_POINT) :: CPV = COLOR_POINT(PV, 3) ! Nested form constructor

  PRINT *, CPV%POINT                  ! Prints 1.0 and 2.0
  PRINT *, CPV%POINT%X, CPV%POINT%Y   ! And this does, too
  PRINT *, CPV%X, CPV%Y               ! And this does, too
```

C.3.2 Allocation with dynamic type (6.3.1)

The following example illustrates the use of allocation with the value and dynamic type of the allocated object given by another object. The example copies a list of objects of any type. It copies the list starting at IN_LIST. After copying, each element of the list starting at LIST_COPY has a polymorphic component, ITEM, for which both the value and type are taken from the ITEM component of the corresponding element of the list starting at IN_LIST.

```
TYPE :: LIST ! A list of anything
  TYPE(LIST), POINTER :: NEXT => NULL()
  CLASS(*), ALLOCATABLE :: ITEM
END TYPE LIST
...
TYPE(LIST), POINTER :: IN_LIST, LIST_COPY => NULL()
TYPE(LIST), POINTER :: IN_WALK, NEW_TAIL
! Copy IN_LIST to LIST_COPY
IF (ASSOCIATED(IN_LIST)) THEN
  IN_WALK => IN_LIST
  ALLOCATE(LIST_COPY)
  NEW_TAIL => LIST_COPY
  DO
    ALLOCATE(NEW_TAIL%ITEM, SOURCE=IN_WALK%ITEM)
    IN_WALK => IN_WALK%NEXT
    IF (.NOT. ASSOCIATED(IN_WALK)) EXIT
    ALLOCATE(NEW_TAIL%NEXT)
    NEW_TAIL => NEW_TAIL%NEXT
  END DO
END IF
```

C.3.3 Pointer allocation and association

The effect of ALLOCATE, DEALLOCATE, NULLIFY, and pointer assignment is that they are interpreted as changing the values in the descriptor that is the pointer. An ALLOCATE is assumed to create space for a suitable object and to "assign" to the pointer the values necessary to describe that space. A NULLIFY breaks the association of the pointer with the space. A DEALLOCATE breaks the association and releases the space. Depending on the implementation, it could be seen as setting a flag in the pointer that indicates whether the values in the descriptor are valid, or it could clear the descriptor values to some (say zero) value indicative of the pointer not pointing to anything. A pointer assignment copies the values necessary to describe the space occupied by the target into the descriptor that is the pointer. Descriptors are copied, values of objects are not.

If PA and PB are both pointers and PB is associated with a target, then

```
PA => PB
```

results in PA being associated with the same target as PB. If PB was disassociated, then PA becomes disassociated.

The standard is specified so that such associations are direct and independent. A subsequent statement

```
PB => D
```

or

```
ALLOCATE (PB)
```

has no effect on the association of PA with its target. A statement

```
DEALLOCATE (PB)
```

deallocates the space that is associated with both PA and PB. PB becomes disassociated, but there is no requirement that the processor make it explicitly recognizable that PA no longer has a target. This leaves PA as a "dangling pointer" to space that has been released. The program shall not use PA again until it becomes associated via pointer assignment or an ALLOCATE statement.

DEALLOCATE may only be used to release space that was created by a previous ALLOCATE. Thus the following is invalid:

```
REAL, TARGET :: T
REAL, POINTER :: P
   ...
P = >  T
DEALLOCATE (P) ! Not allowed: P's target was not allocated
```

The basic principle is that ALLOCATE, NULLIFY, and pointer assignment primarily affect the pointer rather than the target. ALLOCATE creates a new target but, other than breaking its connection with the specified pointer, it has no effect on the old target. Neither NULLIFY nor pointer assignment has any effect on targets. A piece of memory that was allocated and associated with a pointer will become inaccessible to a program if the pointer is nullified or associated with a different target and no other pointer was associated with this piece of memory. Such pieces of memory may be reused by the processor if this is expedient. However, whether such inaccessible memory is in fact reused is entirely processor dependent.

C.4 Section 7 notes

C.4.1 Character assignment

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 (7.4.1.3).

C.4.2 Evaluation of function references

If more than one function reference appears in a statement, they may be executed in any order (subject to a function result being evaluated after the evaluation of its arguments) and their values shall not depend on the order of execution. This lack of dependence on order of evaluation permits parallel execution of the function references (7.1.8.1).

C.4.3 Pointers in expressions

A pointer is basically considered to be like any other variable when it is used as a primary in an expression. If a pointer is used as an operand to an operator that expects a value, the pointer will automatically deliver the value stored in the space described by the pointer, that is, the value of the target object associated with the pointer.

C.4.4 Pointers on the left side of an assignment

A pointer that appears on the left of an intrinsic assignment statement also is dereferenced and is taken to be referring to the space that is its current target. Therefore, the assignment statement specifies the normal copying of the value of the right-hand expression into this target space. All the normal rules of intrinsic assignment hold; the type and type parameters of the expression and the pointer target shall agree and the shapes shall be conformable.

For intrinsic assignment of derived types, nonpointer components are assigned and pointer components are pointer assigned. Dereferencing is applied only to entire scalar objects, not selectively to pointer subobjects.

For example, suppose a type such as

```
TYPE CELL
   INTEGER :: VAL
   TYPE (CELL), POINTER :: NEXT_CELL
END TYPE CELL
```

is defined and objects such as HEAD and CURRENT are declared using

```
TYPE (CELL), TARGET :: HEAD
TYPE (CELL), POINTER :: CURRENT
```

If a linked list has been created and attached to HEAD and the pointer CURRENT has been allocated space, statements such as

```
CURRENT = HEAD
CURRENT = CURRENT % NEXT_CELL
```

cause the contents of the cells referenced on the right to be copied to the cell referred to by CURRENT. In particular, the right-hand side of the second statement causes the pointer component in the cell, CURRENT, to be selected. This pointer is dereferenced because it is in an expression context to produce the target's integer value and a pointer to a cell that is in the target's NEXT_CELL component. The left-hand side causes the pointer CURRENT to be dereferenced to produce its present target, namely space to hold a cell (an integer and a cell pointer). The integer value on the right is copied to the integer space on the left and the pointer component is pointer assigned (the descriptor on the right is copied into the space for a descriptor on the left). When a statement such as

```
CURRENT => CURRENT % NEXT_CELL
```

is executed, the descriptor value in CURRENT % NEXT_CELL is copied to the descriptor named CURRENT. In this case, CURRENT is made to point at a different target.

In the intrinsic assignment statement, the space associated with the current pointer does not change but the values stored in that space do. In the pointer assignment, the current pointer is made to associate with different space. Using the intrinsic assignment causes a linked list of cells to be moved up through the current "window"; the pointer assignment causes the current pointer to be moved down through the list of cells.

C.4.5 An example of a FORALL construct containing a WHERE construct

```
INTEGER :: A(5,5)
...
FORALL (I = 1:5)
   WHERE (A(I,:) == 0)
```

```
      A(:,I) = I
   ELSEWHERE (A(I,:) > 2)
      A(I,:) = 6
   END WHERE
END FORALL
```

If prior to execution of the FORALL, A has the value

```
A =      1  0  0  0  0
         2  1  1  1  0
         1  2  2  0  2
         2  1  0  2  3
         1  0  0  0  0
```

After execution of the assignment statements following the WHERE statement A has the value A'. The mask created from row one is used to mask the assignments to column one; the mask from row two is used to mask assignments to column two; etc.

```
A' =     1  0  0  0  0
         1  1  1  1  5
         1  2  2  4  5
         1  1  3  2  5
         1  2  0  0  5
```

The masks created for assignments following the ELSEWHERE statement are

```
.NOT. (A(I,:)  == 0) .AND. (A'(I,:)  > 2)
```

Thus the only elements affected by the assignments following the ELSEWHERE statement are A(3, 5) and A(4, 5). After execution of the FORALL construct, A has the value

```
A =      1  0  0  0  0
         1  1  1  1  5
         1  2  2  4  6
         1  1  3  2  6
         1  2  0  0  5
```

C.4.6 Examples of FORALL statements

Example 1:

```
FORALL (J=1:M, K=1:N) X(K, J) = Y(J, K)
FORALL (K=1:N) X(K, 1:M) = Y(1:M, K)
```

These statements both copy columns 1 through N of array Y into rows 1 through N of array X. They are equivalent to

```
X(1:N, 1:M) = TRANSPOSE (Y(1:M, 1:N) )
```

Example 2:

The following FORALL statement computes five partial sums of subarrays of J.

```
J = (/ 1, 2, 3, 4, 5 /)
FORALL (K = 1:5) J(K) = SUM (J(1:K) )
```

SUM is allowed in a FORALL because intrinsic functions are pure (12.6). After execution of the FORALL statement, J = (/ 1, 3, 6, 10, 15 /).

Example 3:

```
FORALL (I = 2:N-1) X(I) = (X(I-1) + 2*X(I) + X(I+1) ) / 4
```

has the same effect as

```
X(2:N-1) = (X(1:N-2) + 2*X(2:N-1) + X(3:N) ) / 4
```

C.5 Section 8 notes

C.5.1 Loop control

Fortran provides several forms of loop control:

(1) With an iteration count and a DO variable. This is the classic Fortran DO loop.
(2) Test a logical condition before each execution of the loop (DO WHILE).
(3) DO "forever".

C.5.2 The CASE construct

At most one case block is selected for execution within a CASE construct, and there is no fall-through from one block into another block within a CASE construct. Thus there is no requirement for the user to exit explicitly from a block.

C.5.3 Examples of DO constructs

The following are all valid examples of block DO constructs.

Example 1:

```
      SUM = 0.0
      READ (IUN) N
      OUTER: DO L = 1, N           ! A DO with a construct name
         READ (IUN) IQUAL, M, ARRAY (1:M)
         IF (IQUAL < IQUAL_MIN) CYCLE OUTER    ! Skip inner loop
         INNER: DO 40 I = 1, M     ! A DO with a label and a name
            CALL CALCULATE (ARRAY (I), RESULT)
            IF (RESULT < 0.0) CYCLE
            SUM = SUM + RESULT
            IF (SUM > SUM_MAX) EXIT OUTER
   40    END DO INNER
      END DO OUTER
```

The outer loop has an iteration count of MAX (N, 0), and will execute that number of times or until SUM exceeds SUM_MAX, in which case the EXIT OUTER statement terminates both loops. The inner loop is skipped by the first CYCLE statement if the quality flag, IQUAL, is too low. If CALCULATE returns a negative RESULT, the second CYCLE statement prevents it from being summed. Both loops have construct names and the inner loop also has a label. A construct name is required in the EXIT statement in order to terminate both loops, but is optional in the CYCLE statements because each belongs to its innermost loop.

Example 2:

```
      N = 0
      DO 50, I = 1, 10
         J = I
         DO K = 1, 5
            L = K
            N = N + 1  ! This statement executes 50 times
         END DO        ! Nonlabeled DO inside a labeled DO
   50 CONTINUE
```

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 60, K = 5, 1  ! This inner loop is never executed
            L = K
            N = N + 1
   60    CONTINUE         ! Labeled DO inside a nonlabeled DO
      END DO
```

After execution of the above program fragment, I = 11, J = 10, K = 5, N = 0, and L is not defined by these statements.

The following are all valid examples of nonblock DO constructs:

Example 4:

```
      DO 70
         READ (IUN, '(1X, G14.7)', IOSTAT = IOS) X
         IF (IOS /= 0) EXIT
         IF (X < 0.) GOTO 70
         CALL SUBA (X)
         CALL SUBB (X)
             ...
         CALL SUBY (X)
         CYCLE
   70    CALL SUBNEG (X)  ! SUBNEG called only when X < 0.
```

This is not a block DO construct because it ends with a statement other than END DO or CONTINUE. The loop will continue to execute until an end-of-file condition or input/output error occurs.

Example 5:

```
      SUM = 0.0
      READ (IUN) N
      DO 80, L = 1, N
         READ (IUN) IQUAL, M, ARRAY (1:M)
         IF (IQUAL < IQUAL_MIN) M = 0  ! Skip inner loop
         DO 80 I = 1, M
            CALL CALCULATE (ARRAY (I), RESULT)
            IF (RESULT < 0.) CYCLE
            SUM = SUM + RESULT
            IF (SUM > SUM_MAX) GOTO 81
   80    CONTINUE  ! This CONTINUE is shared by both loops
   81 CONTINUE
```

This example is similar to Example 1 above, except that the two loops are not block DO constructs because they share the CONTINUE statement with the label 80. The terminal construct of the outer DO is the entire inner DO construct. The inner loop is skipped by forcing M to zero. If SUM grows too large, both loops are terminated by branching to the CONTINUE statement labeled 81. The CYCLE statement in the inner loop is used to skip negative values of RESULT.

Example 6:

```
      N = 0
      DO 100 I = 1, 10
         J = I
         DO 100 K = 1, 5
            L = K
  100       N = N + 1  ! This statement executes 50 times
```

In this example, the two loops share an assignment statement. After execution of this program fragment, I = 11, J = 10, K = 6, L = 5, and N = 50.

Example 7:

```
      N = 0
      DO 200 I = 1, 10
         J = I
         DO 200 K = 5, 1  ! This inner loop is never executed
            L = K
  200       N = N + 1
```

This example is very similar to the previous one, except that the inner loop is never executed. After execution of this program fragment, I = 11, J = 10, K = 5, N = 0, and L is not defined by these statements.

C.5.4 Examples of invalid DO constructs

The following are all examples of invalid skeleton DO constructs:

Example 1:

```
DO I = 1, 10
   ...
END DO LOOP   ! No matching construct name
```

Example 2:

```
LOOP: DO 1000 I = 1, 10   ! No matching construct name
   ...
1000  CONTINUE
```

Example 3:

```
LOOP1: DO
   ...
END DO LOOP2   ! Construct names don't match
```

Example 4:

```
DO I = 1, 10   ! Label required or ...
   ...
1010 CONTINUE  ! ... END DO required
```

Example 5:

```
DO 1020 I = 1, 10
   ...
1021 END DO        ! Labels don't match
```

Example 6:

```
FIRST: DO I = 1, 10
   SECOND: DO J = 1, 5
      ...
   END DO FIRST   ! Improperly nested DOs
END DO SECOND
```

C.6 Section 9 notes

C.6.1 External files (9.2)

This standard accommodates, but does not require, file cataloging. To do this, several concepts are introduced.

C.6.1.1 File connection (9.4)

Before any input/output may be performed on a file, it shall 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, or that any other method of implementation has been used. Connection means that (barring some other fault) a READ or WRITE statement may be executed on the unit, hence on the file. Without a connection, a READ or WRITE statement shall not be executed.

C.6.1.2 File existence (9.2.1)

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 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 program because of security, because they are already in use by another 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 a 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 file on a reel of tape, not known to the processor

Means are provided to create, delete, connect, and disconnect files.

C.6.1.3 File names (9.4.5.8)

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 at 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 a 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.

C.6.1.4 File access (9.2.2)

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 might 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.

C.6.2 Nonadvancing input/output (9.2.3.1)

Data transfer statements affect the positioning of an external file. In FORTRAN 77, if no error or end-of-file condition exists, the file is positioned after the record just read or written and that record becomes the preceding record. This standard contains the record positioning ADVANCE= specifier in a data transfer statement that provides the capability of maintaining a position within the current record from one formatted data transfer statement to the next data transfer statement. The value NO provides this capability. The value YES positions the file after the record just read or written. The default is YES.

The tab edit descriptor and the slash are still appropriate for use with this type of record access but the tab will not reposition before the left tab limit.

A BACKSPACE of a file that is positioned within a record causes the specified unit to be positioned before the current record.

If the last data transfer statement was WRITE and the file is positioned within a record, the file will be positioned implicitly after the current record before an ENDFILE record is written to the file, that is, a REWIND, BACKSPACE, or ENDFILE statement following a nonadvancing WRITE statement causes the file to be positioned at the end of the current output record before the endfile record is written to the file.

This standard provides a SIZE= specifier to be used with nonadvancing data transfer statements. The variable in the SIZE= specifier will contain the count of the number of characters that make up the sequence of values read by the data edit descriptors in this input statement.

The count is especially helpful if there is only one list item in the input list because it will contain the number of characters that were present for the item.

The EOR= specifier is provided to indicate when an end-of-record condition has been encountered during a nonadvancing data transfer statement. The end-of-record condition is not an error condition. If this specifier is present, the current input list item that required more characters than the record contained will be padded with blanks if PAD= 'YES' is in effect. This means that the input list item was successfully completed. The file will then be positioned after the current record. The IOSTAT= specifier, if present, will be defined with the value of the named constant IOSTAT_EOR from the ISO_FORTRAN_ENV module and the data transfer statement will be terminated. Program execution will continue with the statement specified in the EOR= specifier. The EOR= specifier gives the capability of taking control of execution when the end-of-record has been found. The *do-variable*s in *io-implied-do*s retain their last defined value and any remaining items in the *input-item-list* retain their definition status when an end-of-record condition occurs. The SIZE= specifier, if present, will contain the number of characters read with the data edit descriptors during this READ statement.

For nonadvancing input, the processor is not required to read partial records. The processor may read the entire record into an internal buffer and make successive portions of the record available to successive input statements.

In an implementation of nonadvancing input/output in which a nonadvancing write to a terminal device causes immediate display of the output, such a write can be used as a mechanism to output a prompt. In this case, the statement

```
WRITE (*, FMT='(A)', ADVANCE='NO') 'CONTINUE?(Y/N): '
```

would result in the prompt

```
CONTINUE?(Y/N):
```

being displayed with no subsequent line feed.

The response, which might be read by a statement of the form

```
READ (*, FMT='(A)') ANSWER
```

can then be entered on the same line as the prompt as in

```
CONTINUE?(Y/N): Y
```

The standard does not require that an implementation of nonadvancing input/output operate in this manner. For example, an implementation of nonadvancing output in which the display of the output is deferred until the current record is complete is also standard conforming. Such an implementation will not, however, allow a prompting mechanism of this kind to operate.

C.6.3 Asynchronous input/output

Rather than limit support for asynchronous input/output to what has been traditionally provided by facilities such as BUFFERIN/BUFFEROUT, this standard builds upon existing Fortran syntax. This permits alternative approaches for implementing asynchronous input/output, and simplifies the task of adapting existing standard conforming programs to use asynchronous input/output.

Not all processors will actually perform input/output asynchronously, nor will every processor that does be able to handle data transfer statements with complicated input/output item lists in an asynchronous manner. Such processors can still be standard conforming. Hopefully, the documentation for each Fortran processor will describe when, if ever, input/output will be performed asynchronously.

This standard allows for at least two different conceptual models for asynchronous input/output.

Model 1: the processor will perform asynchronous input/output when the item list is simple (perhaps one contiguous named array) and the input/output is unformatted. The implementation cost is reduced, and this is the scenario most likely to be beneficial on traditional "big-iron" machines.

Model 2: The processor is free to do any of the following:

(1) on output, create a buffer inside the input/output library, completely formatted, and then start an asynchronous write of the buffer, and immediately return to the next statement in the program. The processor is free to wait for previously issued WRITEs, or not, or

(2) pass the input/output list addresses to another processor/process, which will process the list items independently of the processor that executes the user's code. The addresses of the list items must be computed before the asynchronous READ/WRITE statement completes. There is still an ordering requirement on list item processing to handle things like READ (...) N,(a(i),i=1,N).

The standard allows a user to issue a large number of asynchronous input/output requests, without waiting for any of them to complete, and then wait for any or all of them. It may be impossible, and undesirable to keep track of each of these input/output requests individually.

It is not necessary for all requests to be tracked by the runtime library. If an ID= specifier does not appear in on a READ or WRITE statement, the runtime is free to forget about this particular request once it has successfully completed. If it gets an ERR or END condition, the processor is free to report this during any input/output operation to that unit. When an ID= specifier is present, the processor's runtime input/output library is required to keep track of any END or ERR conditions for that particular input/output request. However, if the input/output request succeeds without any exceptional conditions occurring, then the runtime can forget that ID= value if it wishes. Typically, a runtime might only keep track of the last request made, or perhaps a very few. Then, when a user WAITs for a particular request, either the library knows about it (and does the right thing with respect to error handling, etc.), or will assume it is one of those requests that successfully completed and was forgotten about (and will just return without signaling any end or error conditions). It is incumbent on the user to pass valid ID=

values. There is no requirement on the processor to detect invalid ID= values. There is of course, a processor dependent limit on how many outstanding input/output requests that generate an end or error condition can be handled before the processor runs out of memory to keep track of such conditions. The restrictions on the SIZE= variables are designed to allow the processor to update such variables at any time (after the request has been processed, but before the WAIT operation), and then forget about them. That's why there is no SIZE= specifier allowed in the various WAIT operations. Only exceptional conditions (errors or ends of files) are expected to be tracked by individual request by the runtime, and then only if an ID= specifier was present. The END= and EOR= specifiers have not been added to all statements that can be WAIT operations. Instead, the IOSTAT variable will have to be queried after a WAIT operation to handle this situation. This choice was made because we expect the WAIT statement to be the usual method of waiting for input/output to complete (and WAIT does support the END= and EOR= specifiers). This particular choice is philosophical, and was not based on significant technical difficulties.

Note that the requirement to set the IOSTAT variable correctly requires an implementation to remember which input/output requests got an EOR condition, so that a subsequent wait operation will return the correct IOSTAT value. This means there is a processor defined limit on the number of outstanding nonadvancing input/output requests that got an EOR condition (constrained by available memory to keep track of this information, similar to END/ERR conditions).

C.6.4 OPEN statement (9.4.5)

A file may become connected to a unit either by preconnection or by execution of an OPEN statement. Preconnection is performed prior to the beginning of execution of a 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 (9.4.4).

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-dependent default file to the specified unit. (The default file might or might 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 creates the file. With the other two methods, execution of the OPEN statement creates the file.

When an OPEN statement is executed, the unit specified in the OPEN might or might not already be connected to a file. If it is already connected to a file (either through preconnection or by a prior OPEN), then omitting the FILE= specifier in the OPEN statement implies that the file is to remain connected to the unit. Such an OPEN statement may be used to change the values of the blank interpretation mode, decimal edit mode, pad mode, I/O rounding mode, delimiter mode, and sign mode.

If the value of the ACTION= specifier is WRITE, then READ statements shall not refer to this connection. ACTION = 'WRITE' does not restrict positioning by a BACKSPACE statement or positioning specified by the POSITION= specifier with the value APPEND. However, a BACKSPACE statement or an OPEN statement containing POSITION = 'APPEND' may fail if the processor requires reading of the file to achieve the positioning.

The following examples illustrate these rules. In the first example, unit 10 is preconnected to a SCRATCH file; the OPEN statement changes the value of PAD= to YES.

```
CHARACTER (LEN = 20) CH1
WRITE (10, '(A)') 'THIS IS RECORD 1'
OPEN (UNIT = 10, STATUS = 'OLD', PAD = 'YES')
REWIND 10
READ (10, '(A20)') CH1   ! CH1 now has the value
                         ! 'THIS IS RECORD 1    '
```

In the next example, unit 12 is first connected to a file named FRED, with a status of OLD. The second OPEN statement then opens unit 12 again, retaining the connection to the file FRED, but changing the value of the DELIM= specifier to QUOTE.

```
CHARACTER (LEN = 25) CH2, CH3
OPEN (12, FILE = 'FRED', STATUS = 'OLD', DELIM = 'NONE')
CH2 = '''THIS STRING HAS QUOTES.'''
             ! Quotes in string CH2
WRITE (12, *) CH2           ! Written with no delimiters
OPEN (12, DELIM = 'QUOTE')  ! Now quote is the delimiter
REWIND 12
READ (12, *) CH3  ! CH3 now has the value
                  ! 'THIS STRING HAS QUOTES.  '
```

The next example is invalid because it attempts to change the value of the STATUS= specifier.

```
OPEN (10, FILE = 'FRED', STATUS = 'OLD')
WRITE (10, *) A, B, C
OPEN (10, STATUS = 'SCRATCH')   ! Attempts to make FRED
                                ! a SCRATCH file
```

The previous example could be made valid by closing the unit first, as in the next example.

```
OPEN (10, FILE = 'FRED', STATUS = 'OLD')
WRITE (10, *) A, B, C
CLOSE (10)
OPEN (10, STATUS = 'SCRATCH')   ! Opens a different
                                ! SCRATCH file
```

C.6.5 Connection properties (9.4.3)

When a unit becomes connected to a file, either by execution of an OPEN statement or by preconnection, the following connection properties, among others, may be established:

(1) An access method, which is sequential, direct, or stream, is established for the connection (9.4.5.1).

(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

9.2.2) 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) (9.4.5.9).

(3) A record length may be established. If the access method is direct, the connection establishes a record length that specifies the length of each record of the file. An existing file with records that are not all of equal length shall not be connected for direct access.

If the access method is sequential, records of varying lengths are permitted. In this case, the record length established specifies the maximum length of a record in the file (9.4.5.12).

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 a program. Some processors may require no job control action prior to execution. This standard enables processors to perform dynamic open, close, or file creation operations, but it does not require such capabilities of the processor.

The meaning of "open" in contexts other than 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 might or might 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.

C.6.6 CLOSE statement (9.4.6)

Similarly, the actions of dismounting a tape, protection, etc. of a "close" may be implicit at the end of a run. The CLOSE statement might or might 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 a unit 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 shall 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.

C.7 Section 10 notes

C.7.1 Number of records (10.3, 10.4, 10.7.2)

The number of records read by an explicitly formatted advancing 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 an explicitly formatted advancing 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.

C.7.2 List-directed input (10.9.1)

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:

```
record 1:  b1b,4bbbbb
record 2:  ,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:

```
record 1:  'bbbbbbbb'
record 2:  'QXY'b'Z'
```

Result: A = 'bbbbbbbb', B = 'Q'

Explanation: In the first record, the rightmost apostrophe is interpreted as delimiting the constant (it cannot be the first of a pair of embedded apostrophes representing a single apostrophe because this would involve the prohibited "splitting" of the pair by the end of a record); therefore, A is assigned 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.8 Section 11 notes

C.8.1 Main program and block data program unit (11.1, 11.3)

The name of the main program or of a block data program unit has no explicit use within the Fortran language. It is available for documentation and for possible use by a processor.

A processor may implement an unnamed main program or unnamed block data program unit by assigning it a default name. However, this name shall not conflict with any other global name in a standard-conforming program. This might be done by making the default name one that is not permitted in a standard-conforming program (for example, by including a character not normally allowed in names) or by providing some external mechanism such that for any given program the default name can be changed to one that is otherwise unused.

C.8.2 Dependent compilation (11.2)

This standard, like its predecessors, is intended to permit the implementation of conforming processors in which a program can be broken into multiple units, each of which can be separately translated in preparation for execution. Such processors are commonly described as supporting separate compilation. There is an important difference between the way separate compilation can be implemented under this standard and the way it could be implemented under the FORTRAN 77 standard. Under the FORTRAN 77 standard, any information required to translate a program unit was specified in that program unit. Each translation was thus totally independent of all others. Under this standard, a program unit can use information that was specified in a separate module and thus may be dependent on that module. The implementation of this dependency in a processor may be that the translation of a program unit may depend on the results of translating one or more modules. Processors implementing the dependency this way are commonly described as supporting dependent compilation.

The dependencies involved here are new only in the sense that the Fortran processor is now aware of them. The same information dependencies existed under the FORTRAN 77 standard, but it was the programmer's responsibility to transport the information necessary to resolve them by making redundant specifications of the information in multiple program units. The availability of separate but dependent compilation offers several potential advantages over the redundant textual specification of information:

(1) Specifying information at a single place in the program ensures that different program units using that information will be translated consistently. Redundant specification leaves the possibility that different information will erroneously be specified. Even if an INCLUDE line is used to ensure that the text of the specifications is identical in all involved program units, the presence of other specifications (for example, an IMPLICIT statement) may change the interpretation of that text.

(2) During the revision of a program, it is possible for a processor to assist in determining whether different program units have been translated using different (incompatible) versions of a module, although there is no requirement that a processor provide such assistance. Inconsistencies in redundant textual specification of information, on the other hand, tend to be much more difficult to detect.

(3) Putting information in a module provides a way of packaging it. Without modules, redundant specifications frequently shall be interleaved with other specifications in a program unit, making convenient packaging of such information difficult.

(4) Because a processor may be implemented such that the specifications in a module are translated once and then repeatedly referenced, there is the potential for greater efficiency than when the processor shall translate redundant specifications of information in multiple program units.

The exact meaning of the requirement that the public portions of a module be available at the time of reference is processor dependent. For example, a processor could consider a module to be available only after it has been compiled and require that if the module has been compiled separately, the result of that compilation shall be identified to the compiler when compiling program units that use it.

C.8.2.1 USE statement and dependent compilation (11.2.1)

Another benefit of the USE statement is its enhanced facilities for name management. If one needs to use only selected entities in a module, one can do so without having to worry about the names of all the other entities in that module. If one needs to use two different modules that happen to contain entities with the same name, there are several ways to deal with the conflict. If none of the entities with the same name are to be used, they can simply be ignored. If the name happens to refer to the same entity in both modules (for example, if both modules obtained it from a third module), then there is no confusion about what the name denotes and the name can be freely used. If the entities are different and one or both is to be used, the local renaming facility in the USE statement makes it possible to give those entities different names in the program unit containing the USE statements.

A benefit of using the ONLY specifier consistently, as compared to USE without it, is that the module from which each accessed entity is accessed is explicitly specified in each program unit. This means that one need not search other program units to find where each one is defined. This reduces maintenance costs.

A typical implementation of dependent but separate compilation may involve storing the result of translating a module in a file (or file element) whose name is derived from the name of the module. Note, however, that the name of a module is limited only by the Fortran rules and not by the names allowed in the file system. Thus the processor may have to provide a mapping between Fortran names and file system names.

The result of translating a module could reasonably either contain only the information textually specified in the module (with "pointers" to information originally textually specified in other modules) or contain all information specified in the module (including copies of information originally specified in other modules). Although the former approach would appear to save on storage space, the latter approach can greatly simplify the logic necessary to process a USE statement and can avoid the necessity of imposing a limit on the logical "nesting" of modules via the USE statement.

Variables declared in a module retain their definition status on much the same basis as variables in a common block. That is, saved variables retain their definition status throughout the execution of a program, while variables that are not saved retain their definition status only during the execution of scoping units that reference the module. In some cases, it may be appropriate to put a USE statement such as

```
USE MY_MODULE, ONLY:
```

in a scoping unit in order to assure that other procedures that it references can communicate through the module. In such a case, the scoping unit would not access any entities from the module, but the variables not saved in the module would retain their definition status throughout the execution of the scoping unit.

There is an increased potential for undetected errors in a scoping unit that uses both implicit typing and the USE statement. For example, in the program fragment

```
SUBROUTINE SUB
   USE MY_MODULE
   IMPLICIT INTEGER (I-N), REAL (A-H, O-Z)
   X = F (B)
   A = G (X) + H (X + 1)
END SUBROUTINE SUB
```

X could be either an implicitly typed real variable or a variable obtained from the module MY_MODULE and might change from one to the other because of changes in MY_MODULE unrelated to the action performed by SUB. Logic errors resulting from this kind of situation can be extremely difficult to locate. Thus, the use of these features together is discouraged.

C.8.2.2 Accessibility attributes

The PUBLIC and PRIVATE attributes, which can be declared only in modules, divide the entities in a module into those that are actually relevant to a scoping unit referencing the module and those that are not. This information may be used to improve the performance of a Fortran processor. For example, it may be possible to discard much of the information about the private entities once a module has been translated, thus saving on both storage and the time to search it. Similarly, it may be possible to recognize that two versions of a module differ only in the private entities they contain and avoid retranslating program units that use that module when switching from one version of the module to the other.

C.8.3 Examples of the use of modules

C.8.3.1 Identical common blocks

A common block and all its associated specification statements may be placed in a module named, for example, MY_COMMON and accessed by a USE statement of the form

```
USE MY_COMMON
```

that accesses the whole module without any renaming. This ensures that all instances of the common block are identical. Module MY_COMMON could contain more than one common block.

C.8.3.2 Global data

A module may contain only data objects, for example:

```
MODULE DATA_MODULE
   SAVE
   REAL A (10), B, C (20,20)
   INTEGER :: I=0
   INTEGER, PARAMETER :: J=10
   COMPLEX D (J,J)
END MODULE DATA_MODULE
```

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, AMODULE => A, DMODULE => D
```

C.8.3.3 Derived types

A derived type may be defined in a module and accessed in a number of program units. For example:

```
MODULE SPARSE
   TYPE NONZERO
      REAL A
      INTEGER  I, J
   END TYPE NONZERO
END MODULE SPARSE
```

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.

C.8.3.4 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 (*, *)  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 subprograms need access to the work arrays, and all such subprograms would contain the statement

```
USE WORK_ARRAYS
```

C.8.3.5 Procedure libraries

Interface bodies for external procedures in a library may be gathered into a module. This permits the use of argument keywords and optional arguments, and allows static checking of the references. Different versions may be constructed for different applications, using argument 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 (:, :)
         ! The SIZE in the next statement is an intrinsic function
         REAL, DIMENSION (SIZE (X, 2)) :: A, F
         INTEGER FLAG
      END SUBROUTINE LLS
         ...
   END INTERFACE
      ...
END MODULE LIBRARY_LLS
```

This module allows the subroutine LLS to be invoked:

```
USE LIBRARY_LLS
   ...
CALL LLS (X = ABC, A = D, F = XX, FLAG = IFLAG)
   ...
```

C.8.3.6 Operator extensions

In order to extend an intrinsic operator symbol to have an additional meaning, an interface block specifying that operator symbol in the OPERATOR option of the INTERFACE statement may be placed in a module.

For example, // may be extended to perform concatenation of two derived-type objects serving as varying length character strings and + may be extended to specify matrix addition for type MATRIX or interval arithmetic addition for type INTERVAL.

A module might contain several such interface blocks. An operator may be defined by an external function (either in Fortran or some other language) and its procedure interface placed in the module.

C.8.3.7 Data abstraction

In addition to providing a portable means of avoiding the redundant specification of information in multiple program units, a module provides a convenient means of "packaging" related entities, such as the definitions of the representation and operations of an abstract data type. The following example of a module defines a data abstraction for a SET type where the elements of each set are of type integer. The standard set operations of UNION, INTERSECTION, and DIFFERENCE are provided. The CARDINALITY function returns the cardinality of (number of elements in) its set argument. Two functions returning logical values are included, ELEMENT and SUBSET. ELEMENT defines the operator .IN. and SUBSET extends the operator <=. 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 SETF 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. In this SET implementation, set data objects have a maximum cardinality of 200.

```
MODULE INTEGER_SETS
! This module is intended to illustrate use of the module facility
! to define a new type, along with suitable operators.

INTEGER, PARAMETER :: MAX_SET_CARD = 200

TYPE SET                               ! Define SET type
   PRIVATE
   INTEGER CARD
   INTEGER ELEMENT (MAX_SET_CARD)
END TYPE SET

INTERFACE OPERATOR (.IN.)
   MODULE PROCEDURE ELEMENT
END INTERFACE OPERATOR (.IN.)

INTERFACE OPERATOR (<=)
   MODULE PROCEDURE SUBSET
END INTERFACE OPERATOR (<=)

INTERFACE OPERATOR (+)
   MODULE PROCEDURE UNION
END INTERFACE OPERATOR (+)

INTERFACE OPERATOR (-)
   MODULE PROCEDURE DIFFERENCE
END INTERFACE OPERATOR (-)

INTERFACE OPERATOR (*)
   MODULE PROCEDURE INTERSECTION
END INTERFACE OPERATOR (*)

CONTAINS

INTEGER FUNCTION CARDINALITY (A)   ! Returns cardinality of set A
   TYPE (SET), INTENT (IN) :: A
   CARDINALITY = A % CARD
END FUNCTION CARDINALITY

LOGICAL FUNCTION ELEMENT (X, A)           ! Determines if
```

```
   INTEGER, INTENT(IN) :: X               ! element X is in set A
   TYPE (SET), INTENT(IN) :: A
   ELEMENT = ANY (A % ELEMENT (1 : A % CARD) == X)
END FUNCTION ELEMENT

FUNCTION UNION (A, B)                     ! Union of sets A and B
   TYPE (SET) UNION
   TYPE (SET), INTENT(IN) :: A, B
   INTEGER J
   UNION = A
   DO J = 1, B % CARD
      IF (.NOT. (B % ELEMENT (J) .IN. A)) THEN
         IF (UNION % CARD < MAX_SET_CARD) THEN
            UNION % CARD = UNION % CARD + 1
            UNION % ELEMENT (UNION % CARD) = &
                  B % ELEMENT (J)
         ELSE
            ! Maximum set size exceeded . . .
         END IF
      END IF
   END DO
END FUNCTION UNION

FUNCTION DIFFERENCE (A, B)           ! Difference of sets A and B
   TYPE (SET) DIFFERENCE
   TYPE (SET), INTENT(IN) :: A, B
   INTEGER J, X
   DIFFERENCE % CARD = 0             ! The empty set
   DO J = 1, A % CARD
      X = A % ELEMENT (J)
      IF (.NOT. (X .IN. B)) DIFFERENCE = DIFFERENCE + SET (1, X)
   END DO
END FUNCTION DIFFERENCE

FUNCTION INTERSECTION (A, B)       ! Intersection of sets A and B
   TYPE (SET) INTERSECTION
   TYPE (SET), INTENT(IN) :: A, B
   INTERSECTION = A - (A - B)
END FUNCTION INTERSECTION

LOGICAL FUNCTION SUBSET (A, B)          ! Determines if set A is
   TYPE (SET), INTENT(IN) :: A, B       ! a subset of set B
   INTEGER I
   SUBSET = A % CARD <= B % CARD
   IF (.NOT. SUBSET) RETURN             ! For efficiency
```

```
   DO I = 1, A % CARD
      SUBSET = SUBSET .AND. (A % ELEMENT (I) .IN. B)
   END DO
END FUNCTION SUBSET

TYPE (SET) FUNCTION SETF (V)     ! Transfer function between a vector
   INTEGER V (:)                 ! of elements and a set of elements
   INTEGER J                     ! removing duplicate elements
   SETF % CARD = 0
   DO J = 1, SIZE (V)
      IF (.NOT. (V (J) .IN. SETF)) THEN
         IF (SETF % CARD < MAX_SET_CARD) THEN
            SETF % CARD = SETF % CARD + 1
            SETF % ELEMENT (SETF % CARD) = V (J)
         ELSE
            ! Maximum set size exceeded . . .
         END IF
      END IF
   END DO
END FUNCTION SETF

FUNCTION VECTOR (A)              ! Transfer the values of set A
   TYPE (SET), INTENT (IN) :: A  ! into a vector in ascending order
   INTEGER, POINTER :: VECTOR (:)
   INTEGER I, J, K
   ALLOCATE (VECTOR (A % CARD))
   VECTOR = A % ELEMENT (1 : A % CARD)
   DO I = 1, A % CARD - 1         ! Use a better sort if
      DO J = I + 1, A % CARD      ! A % CARD is large
         IF (VECTOR (I) > VECTOR (J)) THEN
            K = VECTOR (J); VECTOR (J) = VECTOR (I); VECTOR (I) = K
         END IF
      END DO
   END DO
END FUNCTION VECTOR

END MODULE INTEGER_SETS
```

Examples of using INTEGER_SETS (A, B, and C are variables of type SET; X is an integer variable):

```
! Check to see if A has more than 10 elements
IF (CARDINALITY (A) > 10) ...

! Check for X an element of A but not of B
IF (X .IN. (A - B)) ...
```

```
! C is the union of A and the result of B intersected
! with the integers 1 to 100
C = A + B * SETF ((/ (I, I = 1, 100) /))

! Does A have any even numbers in the range 1:100?
IF (CARDINALITY (A * SETF ((/ (I, I = 2, 100, 2) /))) > 0) ...

PRINT *, VECTOR (B) ! Print out the elements of set B, in ascending order
```

C.8.3.8 Public entities renamed

At times it may be necessary to rename entities that are accessed with USE statements. Care should be taken if the referenced modules also contain USE statements.

The following example illustrates renaming features of the USE statement.

```
MODULE J; REAL JX, JY, JZ; END MODULE J
MODULE K
   USE J, ONLY : KX => JX, KY => JY
   ! KX and KY are local names to module K
   REAL KZ       ! KZ is local name to module K
   REAL JZ       ! JZ is local name to module K
END MODULE K
PROGRAM RENAME
   USE J; USE K
   ! Module J's entity JX is accessible under names JX and KX
   ! Module J's entity JY is accessible under names JY and KY
   ! Module K's entity KZ is accessible under name KZ
   ! Module J's entity JZ and K's entity JZ are different entities
   ! and shall not be referenced
   ...
END PROGRAM RENAME
```

C.9 Section 12 notes

C.9.1 Portability problems with external procedures (12.3.2.2)

There is a potential portability problem in a scoping unit that references an external procedure without explicitly declaring it to have the EXTERNAL attribute (5.1.2.6). On a different processor, the name of that procedure may be the name of a nonstandard intrinsic procedure and the processor would be permitted to interpret those procedure references as references to that intrinsic procedure. (On that processor, the program would also be viewed as not conforming to the standard because of the references to the nonstandard intrinsic procedure.) Declaration of the EXTERNAL attribute causes the references to be to the external procedure regardless of the availability of an intrinsic procedure with the same name. Note that declaration of the type of a procedure is not enough to make it external, even if the type is inconsistent with the type of the result of an intrinsic of the same name.

C.9.2 Procedures defined by means other than Fortran (12.5.3)

A processor is not required to provide any means other than Fortran for defining external procedures. Among the means that might be supported are the machine assembly language, other high level languages, the Fortran language extended with nonstandard features, and the Fortran language as supported by another Fortran processor (for example, a previously existing FORTRAN 77 processor).

Procedures defined by means other than Fortran are considered external procedures because their definitions are not in a Fortran program unit and because they are referenced using global names. The use of the term external should not be construed as any kind of restriction on the way in which these procedures may be defined. For example, if the means other than Fortran has its own facilities for internal and external procedures, it is permissible to use them. If the means other than Fortran can create an "internal" procedure with a global name, it is permissible for such an "internal" procedure to be considered by Fortran to be an external procedure. The means other than Fortran for defining external procedures, including any restrictions on the structure for organization of those procedures, are entirely processor dependent.

A Fortran processor may limit its support of procedures defined by means other than Fortran such that these procedures may affect entities in the Fortran environment only on the same basis as procedures written in Fortran. For example, it might prohibit the value of a local variable from being changed by a procedure reference unless that variable were one of the arguments to the procedure.

C.9.3 Procedure interfaces (12.3)

In FORTRAN 77, the interface to an external procedure was always deduced from the form of references to that procedure and any declarations of the procedure name in the referencing program unit. In this standard, features such as argument keywords and optional arguments make it impossible to deduce sufficient information about the dummy arguments from the nature of the actual arguments to be associated with them, and features such as array function results and pointer function results make necessary extensions to the declaration of a procedure that cannot be done in a way that would be analogous with the handling of such declarations in FORTRAN 77. Hence, mechanisms are provided through which all the information about a procedure's interface may be made available in a scoping unit that references it. A procedure whose interface shall be deduced as in FORTRAN 77 is described as having an implicit interface. A procedure whose interface is fully known is described as having an explicit interface.

A scoping unit is allowed to contain an interface body for a procedure that does not exist in the program, provided the procedure described is never referenced or used in any other way. The purpose of this rule is to allow implementations in which the use of a module providing interface bodies describing the interface of every routine in a library would not automatically cause each of those library routines to be a part of the program referencing the module. Instead, only those library procedures actually referenced would be a part of the program. (In implementation terms, the mere presence of an interface body would not generate an external reference in such an implementation.)

C.9.4 Abstract interfaces (12.3) and procedure pointer components (4.5)

This is an example of a library module providing lists of callbacks that the user may register and invoke.

```
MODULE callback_list_module
  !
  ! Type for users to extend with their own data, if they so desire
  !
  TYPE callback_data
```

```
  END TYPE
  !
  ! Abstract interface for the callback procedures
  !
  ABSTRACT INTERFACE
    SUBROUTINE callback_procedure(data)
      IMPORT callback_data
      CLASS(callback_data),OPTIONAL :: data
    END SUBROUTINE
  END INTERFACE
  !
  ! The callback list type.
  !
  TYPE callback_list
    PRIVATE
    CLASS(callback_record),POINTER :: first => NULL()
  END TYPE
  !
  ! Internal: each callback registration creates one of these
  !
  TYPE,PRIVATE :: callback_record
    PROCEDURE(callback_procedure),POINTER,NOPASS :: proc
    CLASS(callback_record),POINTER :: next
    CLASS(callback_data),POINTER :: data => NULL();
  END TYPE
  PRIVATE invoke,forward_invoke
CONTAINS
  !
  ! Register a callback procedure with optional data
  !
  SUBROUTINE register_callback(list, entry, data)
    TYPE(callback_list),INTENT(INOUT) :: list
    PROCEDURE(callback_procedure) :: entry
    CLASS(callback_data),OPTIONAL :: data
    TYPE(callback_record),POINTER :: new,last
    ALLOCATE(new)
    new%proc => entry
    IF (PRESENT(data)) ALLOCATE(new%data,SOURCE=data)
    new%next => list%first
    list%first => new
  END SUBROUTINE
  !
  ! Internal: Invoke a single callback and destroy its record
  !
  SUBROUTINE invoke(callback)
```

```
    TYPE(callback_record),POINTER :: callback
    IF (ASSOCIATED(callback%data) THEN
      CALL callback%proc(list%first%data)
      DEALLOCATE(callback%data)
    ELSE
      CALL callback%proc
    END IF
    DEALLOCATE(callback)
  END SUBROUTINE
  !
  ! Call the procedures in reverse order of registration
  !
  SUBROUTINE invoke_callback_reverse(list)
    TYPE(callback_list),INTENT(INOUT) :: list
    TYPE(callback_record),POINTER :: next,current
    current => list%first
    NULLIFY(list%first)
    DO WHILE (ASSOCIATED(current))
      next => current%next
      CALL invoke(current)
      current => next
    END DO
  END SUBROUTINE
  !
  ! Internal: Forward mode invocation
  !
  RECURSIVE SUBROUTINE forward_invoke(callback)
    IF (ASSOCIATED(callback%next)) CALL forward_invoke(callback%next)
    CALL invoke(callback)
  END SUBROUTINE
  !
  ! Call the procedures in forward order of registration
  !
  SUBROUTINE invoke_callback_forward(list)
    TYPE(callback_list),INTENT(INOUT) :: list
    IF (ASSOCIATED(list%first)) CALL forward_invoke(list%first)
  END SUBROUTINE
END
```

C.9.5 Argument association and evaluation (12.4.1.2)

There is a significant difference between the argument association allowed in this standard and that supported by FORTRAN 77 and FORTRAN 66. In FORTRAN 77 and 66, actual arguments were limited to consecutive storage units. With the exception of assumed length character dummy arguments, the structure imposed on that sequence of storage units was always determined in the invoked procedure and not taken from the actual argument. Thus it was possible to implement FORTRAN 66 and FORTRAN 77 argument association by supplying only the location of the first storage unit (except for character arguments, where the length would also have to be supplied). However, this standard allows arguments that do not reside in consecutive storage locations (for example, an array section), and dummy arguments that assume additional structural information from the actual argument (for example, assumed-shape dummy arguments). Thus, the mechanism to implement the argument association allowed in this standard needs to be more general.

Because there are practical advantages to a processor that can support references to and from procedures defined by a FORTRAN 77 processor, requirements for explicit interfaces make it possible to determine whether a simple (FORTRAN 66/FORTRAN 77) argument association implementation mechanism is sufficient or whether the more general mechanism is necessary (12.3.1.1). Thus a processor can be implemented whose procedures expect the simple mechanism to be used whenever the procedure's interface is one that uses only FORTRAN 77 features and that expects the more general mechanism otherwise (for example, if there are assumed-shape or optional arguments). At the point of reference, the appropriate mechanism can be determined from the interface if it is explicit and can be assumed to be the simple mechanism if it is not. Note that if the simple mechanism is determined to be what the procedure expects, it may be necessary for the processor to allocate consecutive temporary storage for the actual argument, copy the actual argument to the temporary storage, reference the procedure using the temporary storage in place of the actual argument, copy the contents of temporary storage back to the actual argument, and deallocate the temporary storage.

While this is the particular implementation method these rules were designed to support, it is not the only one possible. For example, on some processors, it may be possible to implement the general argument association in such a way that the information involved in FORTRAN 77 argument association may be found in the same places and the "extra" information is placed so it does not disturb a procedure expecting only FORTRAN 77 argument association. With such an implementation, argument association could be translated without regard to whether the interface is explicit or implicit.

The provisions for expression evaluation give the processor considerable flexibility for obtaining expression values in the most efficient way possible. This includes not evaluating or only partially evaluating an operand, for example, if the value of the expression can be determined otherwise (7.1.8.1). This flexibility applies to function argument evaluation, including the order of argument evaluation, delaying argument evaluation, and omitting argument evaluation. A processor may delay the evaluation of an argument in a procedure reference until the execution of the procedure refers to the value of that argument, provided delaying the evaluation of the argument does not otherwise affect the results of the program. The processor may, with similar restrictions, entirely omit the evaluation of an argument not referenced in the execution of the procedure. This gives processors latitude for optimization (for example, for parallel processing).

C.9.6 Pointers and targets as arguments (12.4.1.2)

If a dummy argument is declared to be a pointer, it may be matched only by an actual argument that also is a pointer, and the characteristics of both arguments shall agree. A model for such an association is that descriptor values of the actual pointer are copied to the dummy pointer. If the actual pointer has an associated target, this target becomes accessible via the dummy pointer. If the dummy pointer becomes associated with a different target during execution of the procedure, this target will be accessible via the actual pointer after the procedure completes execution. If the dummy pointer becomes associated with a local target that ceases to exist when the procedure completes, the actual pointer will be left dangling in an undefined state. Such dangling pointers shall not be used.

When execution of a procedure completes, any pointer that remains defined and that is associated with a dummy argument that has the TARGET attribute and is either a scalar or an assumed-shape array, remains associated with the corresponding actual argument if the actual argument has the TARGET attribute and is not an array section with a vector subscript.

```
REAL, POINTER      :: PBEST
REAL, TARGET       :: B (10000)
CALL BEST (PBEST, B)           ! Upon return PBEST is associated
   ...                         ! with the ‘‘best’’ element of B
CONTAINS
  SUBROUTINE BEST (P, A)
    REAL, POINTER, INTENT (OUT)  :: P
    REAL, TARGET, INTENT (IN)    :: A (:)
       ...                     ! Find the ‘‘best’’ element A(I)
    P => A (I)
  RETURN
  END SUBROUTINE BEST
END
```

When procedure BEST completes, the pointer PBEST is associated with an element of B.

An actual argument without the TARGET attribute can become associated with a dummy argument with the TARGET attribute. This permits pointers to become associated with the dummy argument during execution of the procedure that contains the dummy argument. For example:

```
INTEGER LARGE(100,100)
CALL SUB (LARGE)
   ...
CALL SUB ()
CONTAINS
  SUBROUTINE SUB(ARG)
    INTEGER, TARGET, OPTIONAL :: ARG(100,100)
    INTEGER, POINTER, DIMENSION(:,:) :: PARG
    IF (PRESENT(ARG)) THEN
      PARG => ARG
    ELSE
      ALLOCATE (PARG(100,100))
      PARG = 0
    ENDIF
       ...  ! Code with lots of references to PARG
    IF (.NOT. PRESENT(ARG)) DEALLOCATE(PARG)
  END SUBROUTINE SUB
END
```

Within subroutine SUB the pointer PARG is either associated with the dummy argument ARG or it is associated with an allocated target. The bulk of the code can reference PARG without further calls to the PRESENT intrinsic.

C.9.7 Polymorphic Argument Association (12.4.1.3)

The following example illustrates polymorphic argument association rules using the derived types defined in Note 4.54.

```
TYPE(POINT) :: T2
TYPE(COLOR_POINT) :: T3
CLASS(POINT) :: P2
CLASS(COLOR_POINT) :: P3
! Dummy argument is polymorphic and actual argument is of fixed type
SUBROUTINE SUB2 ( X2 ); CLASS(POINT) :: X2; ...
SUBROUTINE SUB3 ( X3 ); CLASS(COLOR_POINT) :: X3; ...

CALL SUB2 ( T2 ) ! Valid -- The declared type of T2 is the same as the
                 !          declared type of X2.
CALL SUB2 ( T3 ) ! Valid -- The declared type of T3 is extended from
                 !          the declared type of X2.
CALL SUB3 ( T2 ) ! Invalid -- The declared type of T2 is neither the
                 !          same as nor extended from the declared type
                 !          type of X3.
CALL SUB3 ( T3 ) ! Valid -- The declared type of T3 is the same as the
                 !          declared type of X3.
! Actual argument is polymorphic and dummy argument is of fixed type
SUBROUTINE TUB2 ( D2 ); TYPE(POINT) :: D2; ...
SUBROUTINE TUB3 ( D3 ); TYPE(COLOR_POINT) :: D3; ...

CALL TUB2 ( P2 ) ! Valid -- The declared type of P2 is the same as the
                 !          declared type of D2.
CALL TUB2 ( P3 ) ! Invalid -- The declared type of P3 differs from the
                 !          declared type of D2.
CALL TUB2 ( P3%POINT ) ! Valid alternative to the above
CALL TUB3 ( P2 ) ! Invalid -- The declared type of P2 differs from the
                 !          declared type of D3.
SELECT TYPE ( P2 ) ! Valid conditional alternative to the above
CLASS IS ( COLOR_POINT ) ! Works if the dynamic type of P2 is the same
  CALL TUB3 ( P2 )       ! as the declared type of D3, or a type
                         ! extended therefrom.
CLASS DEFAULT
                         ! Cannot work if not.
END SELECT
CALL TUB3 ( P3 ) ! Valid -- The declared type of P3 is the same as the
                 !          declared type of D3.
! Both the actual and dummy arguments are of polymorphic type.
CALL SUB2 ( P2 ) ! Valid -- The declared type of P2 is the same as the
                 !          declared type of X2.
CALL SUB2 ( P3 ) ! Valid -- The declared type of P3 is extended from
                 !          the declared type of X2.
CALL SUB3 ( P2 ) ! Invalid -- The declared type of P2 is neither the
                 !          same as nor extended from the declared
                 !          type of X3.
```

```
SELECT TYPE ( P2 ) ! Valid conditional alternative to the above
CLASS IS ( COLOR_POINT ) ! Works if the dynamic type of P2 is the
  CALL SUB3 ( P2 )       ! same as the declared type of X3, or a
                         ! type extended therefrom.
CLASS DEFAULT
                         ! Cannot work if not.
END SELECT
CALL SUB3 ( P3 ) ! Valid -- The declared type of P3 is the same as the
                 !          declared type of X3.
```

C.10 Section 15 notes

C.10.1 Runtime environments

This standard allows programs to contain procedures defined by means other than Fortran. That raises the issues of initialization of and interaction between the runtime environments involved.

Implementations are free to solve these issues as they see fit, provided that:

(1) Heap allocation/deallocation (e.g., (DE)ALLOCATE in a Fortran subprogram and malloc/free in a C function) can be performed without interference.

(2) I/O to and from external files can be performed without interference, as long as procedures defined by different means do not do I/O to/from the same external file.

(3) I/O preconnections exist as required by the respective standards.

(4) Initialized data is initialized according to the respective standards.

C.10.2 Examples of Interoperation between Fortran and C Functions

The following examples illustrate the interoperation of Fortran and C functions. Two examples are shown: one of Fortran calling C, and one of C calling Fortran. In each of the examples, the correspondences of Fortran actual arguments, Fortran dummy arguments, and C formal parameters are described.

C.10.2.1 Example of Fortran calling C

C Function Prototype:

```
int C_Library_Function(void* sendbuf, int sendcount,
         int *recvcounts);
```

Fortran Modules:

```
MODULE FTN_C_1
   USE, INTRINSIC :: ISO_C_BINDING
END MODULE FTN_C_1

MODULE FTN_C_2
   INTERFACE
```

```
            INTEGER (C_INT) FUNCTION C_LIBRARY_FUNCTION &
            (SENDBUF, SENDCOUNT, RECVCOUNTS),    &
            BIND(C, NAME='C_Library_Function')
               USE FTN_C_1
               IMPLICIT NONE
               TYPE (C_PTR), VALUE :: SENDBUF
               INTEGER (C_INT), VALUE :: SENDCOUNT
               TYPE (C_PTR), VALUE :: RECVCOUNTS
            END FUNCTION C_LIBRARY_FUNCTION
         END INTERFACE
      END MODULE FTN_C_2
```

The module FTN_C_2 contains the declaration of the Fortran dummy arguments, which correspond to the C formal parameters. The intrinsic module ISO_C_BINDING is referenced in the module FTN_C_1. The NAME specifier is used in the BIND attribute in order to handle the case-sensitive name change between Fortran and C from 'C_LIBRARY_FUNCTION' to 'C_Library_Function'. See also Note 12.39.

The first C formal parameter is the pointer to void `sendbuf`, which corresponds to the Fortran dummy argument SENDBUF, which has the type C_PTR and the VALUE attribute.

The second C formal parameter is the int `sendcount`, which corresponds to the Fortran dummy argument SENDCOUNT, which has the type INTEGER(C_INT) and the VALUE attribute.

The third C formal parameter is the pointer to int `recvcounts`, which corresponds to the Fortran dummy argument RECVCOUNTS, which has the type C_PTR and the VALUE attribute.

Fortran Calling Sequence:

```
      USE, INTRINSIC :: ISO_C_BINDING, ONLY: C_INT, C_FLOAT, C_LOC
      USE FTN_C_2
      ...
      REAL (C_FLOAT), TARGET  :: SEND(100)
      INTEGER (C_INT)         :: SENDCOUNT
      INTEGER (C_INT), ALLOCATABLE, TARGET :: RECVCOUNTS(100)
      ...
      ALLOCATE( RECVCOUNTS(100) )
      ...
      CALL C_LIBRARY_FUNCTION(C_LOC(SEND), SENDCOUNT, &
      C_LOC(RECVCOUNTS))
      ...
```

The preceding code contains the declaration of the Fortran actual arguments associated with the above-listed Fortran dummy arguments.

The first Fortran actual argument is the address of the first element of the array SEND, which has the type REAL(C_FLOAT) and the TARGET attribute. This address is returned by the intrinsic function C_LOC. This actual argument is associated with the Fortran dummy argument SENDBUF, which has the type C_PTR and the VALUE attribute.

The second Fortran actual argument is SENDCOUNT of type INTEGER(C_INT), which is associated with the Fortran dummy argument SENDCOUNT, which has the type INTEGER(C_INT) and the VALUE attribute.

The third Fortran actual argument is the address of the first element of the allocatable array RECVCOUNTS, with has the type REAL(C_FLOAT) and the TARGET attribute. This address is returned by the intrinsic function C_LOC. This actual argument is associated with the Fortran dummy argument RECVCOUNTS, which has the type C_PTR and the VALUE attribute.

C.10.2.2 Example of C calling Fortran

Fortran Code:

```
SUBROUTINE SIMULATION(ALPHA, BETA, GAMMA, DELTA, ARRAYS), BIND(C)
   USE, INTRINSIC :: ISO_C_BINDING
   IMPLICIT NONE
   INTEGER (C_LONG), VALUE                 :: ALPHA
   REAL (C_DOUBLE), INTENT(INOUT)          :: BETA
   INTEGER (C_LONG), INTENT(OUT)           :: GAMMA
   REAL (C_DOUBLE),DIMENSION(*),INTENT(IN) :: DELTA
   TYPE, BIND(C) :: PASS
     INTEGER (C_INT) :: LENC, LENF
     TYPE (C_PTR)    :: C, F
   END TYPE PASS
   TYPE (PASS), INTENT(INOUT) :: ARRAYS
   REAL (C_FLOAT), ALLOCATABLE, TARGET, SAVE :: ETA(:)
   REAL (C_FLOAT), POINTER :: C_ARRAY(:)
   ...
   ! Associate C_ARRAY with an array allocated in C
   CALL C_F_POINTER (ARRAYS%C, C_ARRAY, (/ARRAYS%LENC/) )
   ...
   ! Allocate an array and make it available in C
   ARRAYS%LENF = 100
   ALLOCATE (ETA(ARRAYS%LENF))
   ARRAYS%F = C_LOC(ETA)
   ...
END SUBROUTINE SIMULATION
```

C Struct Declaration

```
     struct pass {int lenc, lenf; float* f, c};
```

C Function Prototype:

```
     void simulation(long alpha, double *beta, long *gamma,
        double delta[], pass *arrays);
```

C Calling Sequence:

```
simulation(alpha, &beta, &gamma, delta, &arrays);
```

The above-listed Fortran code specifies a subroutine SIMULATION. This subroutine corresponds to the C void function `simulation`.

The Fortran subroutine references the intrinsic module ISO_C_BINDING.

The first Fortran dummy argument of the subroutine is ALPHA, which has the type INTEGER(C_-LONG) and the attribute VALUE. This dummy argument corresponds to the C formal parameter `alpha`, which is a long. The actual C parameter is also a long.

The second Fortran dummy argument of the subroutine is BETA, which has the type REAL(C_-DOUBLE) and the INTENT(INOUT) attribute. This dummy argument corresponds to the C formal parameter `beta`, which is a pointer to double. An address is passed as the actual parameter in the C calling sequence.

The third Fortran dummy argument of the subroutine is GAMMA, which has the type INTEGER(C_-LONG) and the INTENT(OUT) attribute. This dummy argument corresponds to the C formal parameter `gamma`, which is a pointer to long. An address is passed as the actual parameter in the C calling sequence.

The fourth Fortran dummy argument is the assumed-size array DELTA, which has the type REAL (C_DOUBLE) and the attribute INTENT(IN). This dummy argument corresponds to the C formal parameter `delta`, which is a double array. The actual C parameter is also a double array.

The fifth Fortran dummy argument is ARRAYS, which is a structure for accessing an array allocated in C and an array allocated in Fortran. The lengths of these arrays are held in the components LENC and LENF; their C addresses are help in components C and F.

C.10.2.3 Example of calling C functions with non-interoperable data

Many Fortran processors support 16-byte real numbers, which might not be supported by the C processor. Assume a Fortran programmer wants to use a C procedure from a message passing library for an array of these reals. The C prototype of this procedure is

```
void ProcessBuffer(void *buffer, int n_bytes);
```

with the corresponding Fortran interface

```
USE, INTRINSIC :: ISO_C_BINDING

INTERFACE
   SUBROUTINE PROCESS_BUFFER(BUFFER,N_BYTES), BIND(C,NAME="ProcessBuffer")
      IMPORT :: C_PTR, C_INT
      TYPE(C_PTR), VALUE :: BUFFER ! The ‘‘C address’’ of the array buffer
      INTEGER(C_INT), VALUE :: N_BYTES ! Number of bytes in buffer
   END SUBROUTINE PROCESS_BUFFER
END INTERFACE
```

This may be done using C_LOC if the particular Fortran processor specifies that C_LOC returns an appropriate address:

```
REAL(R_QUAD), DIMENSION(:), ALLOCATABLE, TARGET :: QUAD_ARRAY
...
CALL PROCESS_BUFFER(C_LOC(QUAD_ARRAY), INT(16*SIZE(QUAD_ARRAY),C_INT))
   ! One quad real takes 16 bytes on this processor
```

C.10.2.4 Example of opaque communication between C and Fortran

The following example demonstrates how a Fortran processor can make a modern OO random number generator written in Fortran available to a C program:

```
USE, INTRINSIC :: ISO_C_BINDING
   ! Assume this code is inside a module

TYPE RANDOM_STREAM
   ! A (uniform) random number generator (URNG)
CONTAINS
   PROCEDURE(RANDOM_UNIFORM), DEFERRED, PASS(STREAM) :: NEXT
   ! Generates the next number from the stream
END TYPE RANDOM_STREAM

ABSTRACT INTERFACE
   ! Abstract interface of Fortran URNG
   SUBROUTINE RANDOM_UNIFORM(STREAM, NUMBER)
      IMPORT :: RANDOM_STREAM, C_DOUBLE
      CLASS(RANDOM_STREAM), INTENT(INOUT) :: STREAM
      REAL(C_DOUBLE), INTENT(OUT) :: NUMBER
   END SUBROUTINE RANDOM_UNIFORM
END INTERFACE
```

A polymorphic object of base type RANDOM_STREAM is not interoperable with C. However, we can make such a random number generator available to C by packaging it inside another nonpolymorphic, nonparameterized derived type:

```
TYPE :: URNG_STATE ! No BIND(C), as this type is not interoperable
   CLASS(RANDOM_STREAM), ALLOCATABLE :: STREAM
END TYPE URNG_STATE
```

The following two procedures will enable a C program to use our Fortran uniform random number generator:

```
! Initialize a uniform random number generator:
SUBROUTINE INITIALIZE_URNG(STATE_HANDLE, METHOD), &
              BIND(C, NAME="InitializeURNG")
    TYPE(C_PTR), INTENT(OUT) :: STATE_HANDLE
      ! An opaque handle for the URNG
    CHARACTER(C_CHAR), DIMENSION(*), INTENT(IN) :: METHOD
```

```
! The algorithm to be used
```

```
         ...
      I = J      ! Use of variable J from program A
                 ! through host association
   END SUBROUTINE B
END PROGRAM A
```

Example 2:

```
PROGRAM A
   TYPE T
      ...
   END TYPE T
   ...
CONTAINS
   SUBROUTINE B
      IMPLICIT TYPE (T) (C)  ! Refers to type T declared below
                             ! in subroutine B, not type T
                             ! declared above in program A
         ...
      TYPE T
         ...
      END TYPE T
         ...
   END SUBROUTINE B
END PROGRAM A
```

Example 3:

```
PROGRAM Q
   REAL (KIND = 1) :: C
      ...
CONTAINS
   SUBROUTINE R
      REAL (KIND = KIND (C)) :: D  ! Invalid declaration
                                   ! See below
      REAL (KIND = 2) :: C
         ...
   END SUBROUTINE R
END PROGRAM Q
```

In the declaration of D in subroutine R, the use of C would refer to the declaration of C in subroutine R, not program Q. However, it is invalid because the declaration of C is required to occur before it is used in the declaration of D (7.1.7).

C.11.2 Rules ensuring unambiguous generics (16.2.3)

The rules in 16.2.3 are intended to ensure

- that it is possible to reference each specific procedure in the generic collection,
- that for any valid reference to the generic procedure, the determination of the specific procedure referenced is unambiguous, and
- that the determination of the specific procedure referenced can be made before execution of the program begins (during compilation).

Specific procedures are distinguished by fixed properties of their arguments, specifically type, kind type parameters, and rank. A valid reference to one procedure in a generic collection will differ from another because it has an argument that the other cannot accept, because it is missing an argument that the other requires, or because one of these fixed properties is different.

Although the declared type of a data entity is a fixed property, polymorphic variables allow for a limited degree of type mismatch between dummy arguments and actual arguments, so the requirement for distinguishing two dummy arguments is type incompatibility, not merely different types. (This is illustrated in the `BAD6` example later in this note.)

That same limited type mismatch means that two dummy arguments that are not type incompatible can be distinguished on the basis of the values of the kind type parameters they have in common; if one of them has a kind type parameter that the other does not, that is irrelevant in distinguishing them.

Rank is a fixed property, but some forms of array dummy arguments allow rank mismatches when a procedure is referenced by its specific name. In order to allow rank to always be usable in distinguishing generics, such rank mismatches are disallowed for those arguments when the procedure is referenced as part of a generic. Additionally, the fact that elemental procedures can accept array arguments is not taken into account when applying these rules, so apparent ambiguity between elemental and nonelemental procedures is possible; in such cases, the reference is interpreted as being to the nonelemental procedure.

For procedures referenced as operators or defined-assignment, syntactically distinguished arguments are mapped to specific positions in the argument list, so the rule for distinguishing such procedures is that it be possible to distinguish the arguments at one of the argument positions.

For user-defined derived-type input/output procedures, only the `dtv` argument corresponds to something explicitly written in the program, so it is the `dtv` that is required to be distinguished. Because `dtv` arguments are required to be scalar, they cannot differ in rank. Thus this rule effectively involves only type and kind type parameters.

For generic procedures identified by names, the rules are more complicated because optional arguments may be omitted and because arguments may be specified either positionally or by name.

In the special case of type-bound procedures with passed-object dummy arguments, the passed-object argument is syntactically distinguished in the reference, so rule (2) can be applied. The type of passed-object arguments is constrained in ways that prevent passed-object arguments in the same scoping unit from being type incompatible. Thus this rule effectively involves only kind type parameters and rank.

The primary means of distinguishing named generics is rule (3). The most common application of that rule is a single argument satisfying both (3a) and (3b):

```
INTERFACE GOOD1
  FUNCTION F1A(X)
    REAL :: F1A,X
  END FUNCTION F1A
  FUNCTION F1B(X)
    INTEGER :: F1B,X
```

```
      END FUNCTION F1B
    END INTERFACE GOOD1
```

Whether one writes `GOOD1(1.0)` or `GOOD1(X=1.0)`, the reference is to `F1A` because `F1B` would require an integer argument whereas these references provide the real constant 1.0.

This example and those that follow are expressed using interface bodies, with type as the distinguishing property. This was done to make it easier to write and describe the examples. The principles being illustrated are equally applicable when the procedures get their explicit interfaces in some other way or when kind type parameters or rank are the distinguishing property.

Another common variant is the argument that satisfies (3a) and (3b) by being required in one specific and completely missing in the other:

```
    INTERFACE GOOD2
      FUNCTION F2A(X)
        REAL :: F2A,X
      END FUNCTION F2A
      FUNCTION F2B(X,Y)
        COMPLEX :: F2B
        REAL :: X,Y
      END FUNCTION F2B
    END INTERFACE GOOD2
```

Whether one writes `GOOD2(0.0,1.0)`, `GOOD2(0.0,Y=1.0)`, or `GOOD2(Y=1.0,X=0.0)`, the reference is to `F2B`, because `F2A` has no argument in the second position or with the name `Y`. This approach is used as an alternative to optional arguments when one wants a function to have different result type, kind type parameters, or rank, depending on whether the argument is present. In many of the intrinsic functions, the DIM argument works this way.

It is possible to construct cases where different arguments are used to distinguish positionally and by name:

```
    INTERFACE GOOD3
      SUBROUTINE S3A(W,X,Y,Z)
        REAL :: W,Y
        INTEGER :: X,Z
      END SUBROUTINE S3A
      SUBROUTINE S3B(X,W,Z,Y)
        REAL :: W,Z
        INTEGER :: X,Y
      END SUBROUTINE S3B
    END INTERFACE GOOD3
```

If one writes `GOOD3(1.0,2,3.0,4)` to reference `S3A`, then the third and fourth arguments are consistent with a reference to `S3B`, but the first and second are not. If one switches to writing the first two arguments as keyword arguments in order for them to be consistent with a reference to `S3B`, the latter two arguments must also be written as keyword arguments, `GOOD3(X=2,W= 1.0,Z=4,Y=3.0)`, and the named arguments `Y` and `Z` are distinguished.

The ordering requirement in rule (3) is critical:

```
INTERFACE BAD4  ! this interface is invalid !
  SUBROUTINE S4A(W,X,Y,Z)
    REAL :: W,Y
    INTEGER :: X,Z
  END SUBROUTINE S4A
  SUBROUTINE S4B(X,W,Z,Y)
    REAL :: X,Y
    INTEGER :: W,Z
  END SUBROUTINE S4B
END INTERFACE BAD4
```

In this example, the positionally distinguished arguments are Y and Z, and it is W and X that are distinguished by name. In this order it is possible to write BAD4(1.0,2,Y=3.0,Z=4), which is a valid reference for both S4A and S4B.

Rule (1) can be used to distinguish some cases that are not covered by rule (3):

```
INTERFACE GOOD5
  SUBROUTINE S5A(X)
    REAL :: X
  END SUBROUTINE S5A
  SUBROUTINE S5B(Y,X)
    REAL :: Y,X
  END SUBROUTINE S5B
END INTERFACE GOOD5
```

In attempting to apply rule (3), position 2 and name Y are distinguished, but they are in the wrong order, just like the BAD4 example. However, when we try to construct a similarly ambiguous reference, we get GOOD5(1.0,X=2.0), which can't be a reference to S5A because it would be attempting to associate two different actual arguments with the dummy argument X. Rule (3) catches this case by recognizing that S5B requires two real arguments, and S5A cannot possibly accept more than one.

The application of rule (1) becomes more complicated when extensible types are involved. If FRUIT is an extensible type, PEAR and APPLE are extensions of FRUIT, and BOSC is an extension of PEAR, then

```
INTERFACE BAD6  ! this interface is invalid !
  SUBROUTINE S6A(X,Y)
    CLASS(PEAR) :: X,Y
  END SUBROUTINE S6A
  SUBROUTINE S6B(X,Y)
    CLASS(FRUIT) :: X
    CLASS(BOSC) :: Y
  END SUBROUTINE S6B
END INTERFACE BAD6
```

might, at first glance, seem distinguishable this way, but because of the limited type mismatching allowed, `BAD6(A_PEAR,A_BOSC)` is a valid reference to both `S6A` and `S6B`.

It is important to try rule (1) for each type present:

```
INTERFACE GOOD7
  SUBROUTINE S7A(X,Y,Z)
    CLASS(PEAR) :: X,Y,Z
  END SUBROUTINE S7A
  SUBROUTINE S7B(X,Z,W)
    CLASS(FRUIT) :: X
    CLASS(BOSC) :: Z
    CLASS(APPLE),OPTIONAL :: W
  END SUBROUTINE S7B
END INTERFACE GOOD7
```

Looking at the most general type, `S7A` has a minimum and maximum of 3 `FRUIT` arguments, while `S7B` has a minimum of 2 and a maximum of three. Looking at the most specific, `S7A` has a minimum of 0 and a maximum of 3 `BOSC` arguments, while `S7B` has a minimum of 1 and a maximum of 2. However, when we look at the intermediate, `S7A` has a minimum and maximum of 3 `PEAR` arguments, while `S7B` has a minimum of 1 and a maximum of 2. Because `S7A`'s minimum exceeds `S7B`'s maximum, they can be distinguished.

In identifying the minimum number of arguments with a particular set of properties, we exclude optional arguments and test TKR compatibility, so the corresponding actual arguments are required to have those properties. In identifying the maximum number of arguments with those properties, we include the optional arguments and test not distinguishable, so we include actual arguments which could have those properties but are not required to have them.

These rules are sufficient to ensure that references to procedures that meet them are unambiguous, but there remain examples that fail to meet these rules but which can be shown to be unambiguous:

```
INTERFACE BAD8  ! this interface is invalid !
! despite the fact that it is unambiguous !
  SUBROUTINE S8A(X,Y,Z)
    REAL,OPTIONAL :: X
    INTEGER :: Y
    REAL :: Z
  END SUBROUTINE S8A
  SUBROUTINE S8B(X,Z,Y)
    INTEGER,OPTIONAL :: X
    INTEGER :: Z
    REAL :: Y
  END SUBROUTINE S8B
END INTERFACE BAD8
```

This interface fails rule (3) because there are no required arguments that can be distinguished from the positionally corresponding argument, but in order for the mismatch of the optional arguments not to be relevant, the later arguments must be specified as keyword arguments, so distinguishing by name does the trick. This interface is nevertheless invalid so a standard- conforming Fortran processor is not required to do such reasoning. The rules to cover all cases are too complicated to be useful.

In addition to not recognizing distinguishable patterns like the one in BAD8, the rules do not distinguish on the basis of any properties other than type, kind type parameters, and rank:

```
INTERFACE BAD9  ! this interface is invalid !
! despite the fact that it is unambiguous !
  SUBROUTINE S9A(X)
    REAL :: X
  END SUBROUTINE S9A
  SUBROUTINE S9B(X)
    INTERFACE
      FUNCTION X(A)
        REAL :: X,A
      END FUNCTION X
    END INTERFACE
  END SUBROUTINE S9B
  SUBROUTINE S9C(X)
    INTERFACE
      FUNCTION X(A)
        REAL :: X
        INTEGER :: A
      END FUNCTION X
    END INTERFACE
  END SUBROUTINE S9C
END INTERFACE BAD9
```

The real data objects that would be valid arguments for S9A are entirely disjoint from procedures that are valid arguments to S9B and S9C, and the procedures that valid arguments for S9B are disjoint from the procedures that are valid arguments to S9C because the former are required to accept real arguments and the latter integer arguments. Again, this interface is invalid, so a standard-conforming Fortran processor need not examine such properties when deciding whether a generic collection is valid. Again, the rules to cover all cases are too complicated to be useful.

C.12 Array feature notes

C.12.1 Summary of features

This section is a summary of the principal array features.

C.12.1.1 Whole array expressions and assignments (7.4.1.2, 7.4.1.3)

An important feature is that whole array expressions and assignments are 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 arrays and may be generic with scalar versions. All arrays in an expression or across an assignment shall conform; that is, have exactly the same shape (number of dimensions and extents 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.12.1.2 Array sections (2.4.5, 6.2.2.3)

Whenever whole arrays may be used, it is also possible to use subarrays 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 of the fourth dimension. This is an artificial example chosen to illustrate the different forms. Of course, a common use may be to select a row or column of an array, for example:

```
A (:, J)
```

C.12.1.3 WHERE statement (7.4.3)

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 > 0) B = LOG (A)
```

takes the logarithm only for positive components of A and makes assignments only in these positions.

The WHERE statement also has a block form (WHERE construct).

C.12.1.4 Automatic arrays and allocatable variables (5.1, 5.1.2.5.3)

Two features useful for writing modular software are automatic arrays, created on entry to a subprogram and destroyed on return, and allocatable variables, including 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); REAL, ALLOCATABLE :: HEAP (:, :)
```

specify 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 some allocatable variables.

C.12.1.5 Array constructors (4.7)

Arrays, and in particular array constants, may be constructed with array constructors exemplified by:

```
(/ 1.0, 3.0, 7.2 /)
```

which is a rank-one array of size 3,

```
(/ (1.3, 2.7, L = 1, 10), 7.1 /)
```

which is a rank-one array of size 21 and contains the pair of real constants 1.3 and 2.7 repeated 10 times followed by 7.1, and

```
(/ (I, I = 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.12.2 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.12.2.1 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 perform 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 ((/ (K, K = 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:

$$\begin{aligned} \text{A} &= \text{(/ A (1), ..., A (N) /)} \\ \text{X} &= \text{(/ X (1), ..., X (N) /)} \\ \text{COS (X)} &= \text{(/ COS (X (1)), ..., COS (X (N)) /)} \\ \text{A * COS (X)} &= \text{(/ A (1) * COS (X (1)), ..., A (N) * COS (X (N)) /)} \end{aligned}$$

The final scalar result is obtained simply by summing the elements of the last of these arrays. Thus, the processor is dealing with arrays at every step of the calculation.

C.12.2.2 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 by 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 mask required is the logical array expression ABS (A) < 0.01. Note that the stages of evaluation of this expression are:

$$\begin{aligned} \text{A} &= \text{(/ A (1), ..., A (N) /)} \\ \text{ABS (A)} &= \text{(/ ABS (A (1)), ..., ABS (A (N)) /)} \\ \text{ABS (A)} < 0.01 &= \text{(/ ABS (A (1))} < 0.01\text{, ..., ABS (A (N))} < 0.01 \text{ /)} \end{aligned}$$

The conditional Fourier sum we arrive at is:

```
CF = SUM (A * COS (X), MASK = ABS (A) < 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 zero an entire array, 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.12.2.3 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 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 gridpoints where a mere 3 or fewer of the 6 nearest neighbors 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.12.2.3.1 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 function to return an array of random numbers; and
(3) Determining which gridpoints are to be flipped.

C.12.2.3.2 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 THRESHOLD (N, N, N)
The array function needed is:
FUNCTION RAND (N)
REAL RAND (N, N, N)
```

The transition probabilities are specified 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
```

make 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, SHIFT = -1, DIM = 1)  &
      + CSHIFT (ONES, SHIFT =  1, DIM = 1)  &
      + CSHIFT (ONES, SHIFT = -1, DIM = 2)  &
      + CSHIFT (ONES, SHIFT =  1, DIM = 2)  &
      + CSHIFT (ONES, SHIFT = -1, DIM = 3)  &
      + CSHIFT (ONES, SHIFT =  1, DIM = 3)
```

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 FLIPS. The decision to flip will be based on the use of uniformly distributed random numbers from the interval $0 \leq p < 1$. These will be provided at each gridpoint by the array function RAND. 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 assigned 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 = 1.0
WHERE (COUNT == 4) THRESHOLD = P (4)
WHERE (COUNT == 5) THRESHOLD = P (5)
WHERE (COUNT == 6) THRESHOLD = P (6)
```

and the spins that are to be flipped are located by the statement:

```
FLIPS = RAND (N) <= 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.12.2.3.3 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)

   DO I = 1, ITERATIONS
      ONES = 0
      WHERE (ISING)  ONES = 1
      COUNT = CSHIFT (ONES, -1, 1) + CSHIFT (ONES, 1, 1) &
            + CSHIFT (ONES, -1, 2) + CSHIFT (ONES, 1, 2) &
            + CSHIFT (ONES, -1, 3) + CSHIFT (ONES, 1, 3)
      WHERE (.NOT. ISING)  COUNT = 6 - COUNT
      THRESHOLD = 1.0
      WHERE (COUNT == 4)  THRESHOLD = P (4)
      WHERE (COUNT == 5)  THRESHOLD = P (5)
      WHERE (COUNT == 6)  THRESHOLD = P (6)
```

```
      FLIPS = RAND (N) <= THRESHOLD
      WHERE (FLIPS)  ISING = .NOT. ISING
   END DO

CONTAINS
   FUNCTION RAND (N)
      REAL RAND (N, N, N)
      CALL RANDOM_NUMBER (HARVEST = RAND)
      RETURN
   END FUNCTION RAND
END
```

C.12.2.3.4 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 (RAND (N) <= 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.12.3 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.12.3.1 A sum of products

The expression

$$\sum_{j=1}^{N} \prod_{i=1}^{M} a_{ij}$$

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 $\begin{bmatrix} \mathrm{B} & \mathrm{C} & \mathrm{D} \\ \mathrm{E} & \mathrm{F} & \mathrm{G} \end{bmatrix}$ the result of this expression is BE + CF + DG.

C.12.3.2 A product of sums

The expression

$$\prod_{i=1}^{M}\sum_{j=1}^{N} a_{ij}$$

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 $\begin{bmatrix} B & C & D \\ E & F & G \end{bmatrix}$ the result of this expression is (B+C+D)(E+F+G).

C.12.3.3 Addition of selected elements

The expression

$$\sum_{x_i > 0.0} x_i$$

can be formed using the Fortran expression

```
SUM (X, MASK = X > 0.0)
```

The mask locates the positive elements of the array of rank one. If X has the vector value (0.0, –0.1, 0.2, 0.3, 0.2, –0.1, 0.0), the result of this expression is 0.7.

C.12.4 Sum of squared residuals

The expression

$$\sum_{i=1}^{N}(x_i - x_{\text{mean}})^2$$

can be formed using the Fortran statements

```
XMEAN = SUM (X) / SIZE (X)
SS = SUM ((X - XMEAN) ** 2)
```

Thus, SS is the sum of the squared residuals.

C.12.5 Vector norms: infinity-norm and one-norm

The infinity-norm of vector X = (X (1), ..., X(N)) is defined as the largest of the numbers ABS (X(1)), ..., ABS(X(N)) and therefore has the value MAXVAL (ABS(X)).

The one-norm of vector X is defined as the *sum* of the numbers ABS (X (1)), ..., ABS (X (N)) and therefore has the value SUM (ABS (X)).

C.12.6 Matrix norms: infinity-norm and one-norm

The infinity-norm of the matrix A = (A (I, J)) is the largest row-sum of the matrix ABS (A (I, J)) and therefore has the value MAXVAL (SUM (ABS (A), DIM = 2)).

The one-norm of the matrix A = (A (I, J)) is the largest column-sum of the matrix ABS (A (I, J)) and therefore has the value MAXVAL (SUM (ABS (A), DIM = 1)).

C.12.7 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: The scores on 4 tests made by 3 students are 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 the example: [90, 80, 66].

Question: What is the average of all the scores?

Answer: SUM (T) / SIZE (T); in the example: 62.

Question: How many of the scores in the table are above average?

Answer: ABOVE = T > SUM (T) / SIZE (T); N = COUNT (ABOVE); in the example: ABOVE is the logical array (t = true, . = false): $\begin{bmatrix} \mathrm{t} & \mathrm{t} & \mathrm{t} & . \\ \mathrm{t} & . & . & \mathrm{t} \\ \mathrm{t} & . & . & . \end{bmatrix}$ 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 the example: 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 the example, the answer is no.

C.12.8 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; the array feature that 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.

C.12.9 Example of element-by-element computation

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.

The procedure is illustrated by the code to evaluate the three cubic polynomials

$$P(t) = 1 + 2t - 3t^2 + 4t^3$$
$$Q(t) = 2 - 3t + 4t^2 - 5t^3$$
$$R(t) = 3 + 4t - 5t^2 + 6t^3$$

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 $\begin{bmatrix} 1 & 2 & -3 & 4 \\ 2 & -3 & 4 & -5 \\ 3 & 4 & -5 & 6 \end{bmatrix}$.

C.12.10 Bit manipulation and inquiry procedures

The procedures IOR, IAND, NOT, IEOR, ISHFT, ISHFTC, IBITS, MVBITS, BTEST, IBSET, and IBCLR are defined by MIL-STD 1753 for scalar arguments and are extended in this standard to accept array arguments and to return array results.

Annex D

(Informative)

Syntax rules

D.1 Extract of all syntax rules

Section 1:

R101	*xyz-list*	**is**	*xyz* [, *xyz*] ...
R102	*xyz-name*	**is**	*name*
R103	*scalar-xyz*	**is**	*xyz*

C101 (R103) *scalar-xyz* shall be scalar.

Section 2:

R201	*program*	**is**	*program-unit*
			[*program-unit*] ...
R202	*program-unit*	**is**	*main-program*
		or	*external-subprogram*
		or	*module*
		or	*block-data*
R203	*external-subprogram*	**is**	*function-subprogram*
		or	*subroutine-subprogram*
R204	*specification-part*	**is**	[*use-stmt*] ...
			[*import-stmt*] ...
			[*implicit-part*]
			[*declaration-construct*] ...
R205	*implicit-part*	**is**	[*implicit-part-stmt*] ...
			implicit-stmt
R206	*implicit-part-stmt*	**is**	*implicit-stmt*
		or	*parameter-stmt*
		or	*format-stmt*
		or	*entry-stmt*
R207	*declaration-construct*	**is**	*derived-type-def*
		or	*entry-stmt*
		or	*enum-def*
		or	*format-stmt*
		or	*interface-block*
		or	*parameter-stmt*
		or	*procedure-declaration-stmt*
		or	*specification-stmt*
		or	*type-declaration-stmt*
		or	*stmt-function-stmt*
R208	*execution-part*	**is**	*executable-construct*
			[*execution-part-construct*] ...

R209	*execution-part-construct*	**is**	*executable-construct*
		or	*format-stmt*
		or	*entry-stmt*
		or	*data-stmt*
R210	*internal-subprogram-part*	**is**	*contains-stmt*
			internal-subprogram
			[*internal-subprogram*] ...
R211	*internal-subprogram*	**is**	*function-subprogram*
		or	*subroutine-subprogram*
R212	*specification-stmt*	**is**	*access-stmt*
		or	*allocatable-stmt*
		or	*asynchronous-stmt*
		or	*bind-stmt*
		or	*common-stmt*
		or	*data-stmt*
		or	*dimension-stmt*
		or	*equivalence-stmt*
		or	*external-stmt*
		or	*intent-stmt*
		or	*intrinsic-stmt*
		or	*namelist-stmt*
		or	*optional-stmt*
		or	*pointer-stmt*
		or	*protected-stmt*
		or	*save-stmt*
		or	*target-stmt*
		or	*volatile-stmt*
		or	*value-stmt*
R213	*executable-construct*	**is**	*action-stmt*
		or	*associate-construct*
		or	*case-construct*
		or	*do-construct*
		or	*forall-construct*
		or	*if-construct*
		or	*select-type-construct*
		or	*where-construct*
R214	*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	*end-function-stmt*

		or	*end-program-stmt*
		or	*end-subroutine-stmt*
		or	*exit-stmt*
		or	*flush-stmt*
		or	*forall-stmt*
		or	*goto-stmt*
		or	*if-stmt*
		or	*inquire-stmt*
		or	*nullify-stmt*
		or	*open-stmt*
		or	*pointer-assignment-stmt*
		or	*print-stmt*
		or	*read-stmt*
		or	*return-stmt*
		or	*rewind-stmt*
		or	*stop-stmt*
		or	*wait-stmt*
		or	*where-stmt*
		or	*write-stmt*
		or	*arithmetic-if-stmt*
		or	*computed-goto-stmt*

C201 (R208) An *execution-part* shall not contain an *end-function-stmt*, *end-program-stmt*, or *end-subroutine-stmt*.

R215	*keyword*	**is**	*name*

Section 3:

R301	*character*	**is**	*alphanumeric-character*
		or	*special-character*
R302	*alphanumeric-character*	**is**	*letter*
		or	*digit*
		or	*underscore*
R303	*underscore*	**is**	_
R304	*name*	**is**	*letter* [*alphanumeric-character*] ...

C301 (R304) The maximum length of a *name* is 63 characters.

R305	*constant*	**is**	*literal-constant*
		or	*named-constant*
R306	*literal-constant*	**is**	*int-literal-constant*
		or	*real-literal-constant*
		or	*complex-literal-constant*
		or	*logical-literal-constant*
		or	*char-literal-constant*
		or	*boz-literal-constant*
R307	*named-constant*	**is**	*name*
R308	*int-constant*	**is**	*constant*

C302 (R308)*int-constant* shall be of type integer.

R309	*char-constant*	**is**	*constant*

C303 (R309) *char-constant* shall be of type character.

R310	*intrinsic-operator*	**is**	*power-op*
		or	*mult-op*
		or	*add-op*
		or	*concat-op*
		or	*rel-op*
		or	*not-op*
		or	*and-op*
		or	*or-op*
		or	*equiv-op*
R311	*defined-operator*	**is**	*defined-unary-op*
		or	*defined-binary-op*
		or	*extended-intrinsic-op*
R312	*extended-intrinsic-op*	**is**	*intrinsic-operator*
R313	*label*	**is**	*digit* [*digit* [*digit* [*digit* [*digit*]]]]

C304 (R313) At least one digit in a *label* shall be nonzero.

Section 4:

R401	*type-spec*	**is**	*intrinsic-type-spec*
		or	*derived-type-spec*

C401 (R401) The *derived-type-spec* shall not specify an abstract type (4.5.6).

R402	*type-param-value*	**is**	*scalar-int-expr*
		or	*
		or	:

C402 (R402) The *type-param-value* for a kind type parameter shall be an initialization expression.

C403 (R402) A colon may be used as a *type-param-value* only in the declaration of an entity or component that has the POINTER or ALLOCATABLE attribute.

R403	*intrinsic-type-spec*	**is**	INTEGER [*kind-selector*]
		or	REAL [*kind-selector*]
		or	DOUBLE PRECISION
		or	COMPLEX [*kind-selector*]
		or	CHARACTER [*char-selector*]
		or	LOGICAL [*kind-selector*]
R404	*kind-selector*	**is**	([KIND =] *scalar-int-initialization-expr*)

C404 (R404) The value of *scalar-int-initialization-expr* shall be nonnegative and shall specify a representation method that exists on the processor.

R405	*signed-int-literal-constant*	**is**	[*sign*] *int-literal-constant*
R406	*int-literal-constant*	**is**	*digit-string* [_ *kind-param*]
R407	*kind-param*	**is**	*digit-string*
		or	*scalar-int-constant-name*
R408	*signed-digit-string*	**is**	[*sign*] *digit-string*
R409	*digit-string*	**is**	*digit* [*digit*] ...
R410	*sign*	**is**	+
		or	–

C405 (R407) A *scalar-int-constant-name* shall be a named constant of type integer.

C406 (R407) The value of *kind-param* shall be nonnegative.

C407 (R406) The value of *kind-param* shall specify a representation method that exists on the processor.

R411 *boz-literal-constant* **is** *binary-constant*
or *octal-constant*
or *hex-constant*
R412 *binary-constant* **is** B ' *digit* [*digit*] ... '
or B " *digit* [*digit*] ... "
C408 (R412) *digit* shall have one of the values 0 or 1.
R413 *octal-constant* **is** O ' *digit* [*digit*] ... '
or O " *digit* [*digit*] ... "
C409 (R413) *digit* shall have one of the values 0 through 7.
R414 *hex-constant* **is** Z ' *hex-digit* [*hex-digit*] ... '
or Z " *hex-digit* [*hex-digit*] ... "
R415 *hex-digit* **is** *digit*
or A
or B
or C
or D
or E
or F
C410 (R411) A *boz-literal-constant* shall appear only as a *data-stmt-constant* in a DATA statement, as the actual argument associated with the dummy argument A of the numeric intrinsic functions DBLE, REAL or INT, or as the actual argument associated with the X or Y dummy argument of the intrinsic function CMPLX.
R416 *signed-real-literal-constant* **is** [*sign*] *real-literal-constant*
R417 *real-literal-constant* **is** *significand* [*exponent-letter exponent*] [_ *kind-param*]
or *digit-string exponent-letter exponent* [_ *kind-param*]
R418 *significand* **is** *digit-string* . [*digit-string*]
or . *digit-string*
R419 *exponent-letter* **is** E
or D
R420 *exponent* **is** *signed-digit-string*
C411 (R417) If both *kind-param* and *exponent-letter* are present, *exponent-letter* shall be E.
C412 (R417) The value of *kind-param* shall specify an approximation method that exists on the processor.
R421 *complex-literal-constant* **is** (*real-part* , *imag-part*)
R422 *real-part* **is** *signed-int-literal-constant*
or *signed-real-literal-constant*
or *named-constant*
R423 *imag-part* **is** *signed-int-literal-constant*
or *signed-real-literal-constant*
or *named-constant*
C413 (R421) Each named constant in a complex literal constant shall be of type integer or real.
R424 *char-selector* **is** *length-selector*
or (LEN = *type-param-value* , ■
■ KIND = *scalar-int-initialization-expr*)
or (*type-param-value* , ■
■ [KIND =] *scalar-int-initialization-expr*)
or (KIND = *scalar-int-initialization-expr* ■

■ [, LEN =*type-param-value*])

R425 *length-selector* **is** ([LEN =] *type-param-value*)
or * *char-length* [,]

R426 *char-length* **is** (*type-param-value*)
or *scalar-int-literal-constant*

C414 (R424) The value of *scalar-int-initialization-expr*

or *sequence-stmt*

C430 (R429) The same *private-or-sequence* shall not appear more than once in a given *derived-type-def*.

R433 *end-type-stmt* **is** END TYPE [*type-name*]

C431 (R433) If END TYPE is followed by a *type-name*, the *type-name* shall be the same as that in the corresponding *derived-type-stmt*.

R434 *sequence-stmt* **is** SEQUENCE

C432 (R438) If SEQUENCE appears, all derived types specified in component definitions shall be sequence types.

C433 (R429) If SEQUENCE appears, a *type-bound-procedure-part* shall not appear.

R435 *type-param-def-stmt* **is** INTEGER [*kind-selector*] , *type-param-attr-spec* :: ■
■ *type-param-decl-list*

R436 *type-param-decl* **is** *type-param-name* [= *scalar-int-initialization-expr*]

C434 (R435) A *type-param-name* in a *type-param-def-stmt* in a *derived-type-def* shall be one of the *type-param-name*s in the *derived-type-stmt* of that *derived-type-def*.

C435 (R435) Each *type-param-name* in the *derived-type-stmt* in a *derived-type-def* shall appear as a *type-param-name* in a *type-param-def-stmt* in that *derived-type-def*.

R437 *type-param-attr-spec* **is** KIND
or LEN

R438 *component-part* **is** [*component-def-stmt*] ...

R439 *component-def-stmt* **is** *data-component-def-stmt*
or *proc-component-def-stmt*

R440 *data-component-def-stmt* **is** *declaration-type-spec* [[, *component-attr-spec-list*] ::] ■
■ *component-decl-list*

R441 *component-attr-spec* **is** POINTER
or DIMENSION (*component-array-spec*)
or ALLOCATABLE
or *access-spec*

R442 *component-decl* **is** *component-name* [(*component-array-spec*)] ■
■ [* *char-length*] [*component-initialization*]

R443 *component-array-spec* **is** *explicit-shape-spec-list*
or *deferred-shape-spec-list*

R444 *component-initialization* **is** = *initialization-expr*
or => *null-init*

C436 (R440) No *component-attr-spec* shall appear more than once in a given *component-def-stmt*.

C437 (R440) A component declared with the CLASS keyword (5.1.1.2) shall have the ALLOCATABLE or POINTER attribute.

C438 (R440) If the POINTER attribute is not specified for a component, the *declaration-type-spec* in the *component-def-stmt* shall specify an intrinsic type or a previously defined derived type.

C439 (R440) If the POINTER attribute is specified for a component, the *declaration-type-spec* in the *component-def-stmt* shall specify an intrinsic type or any accessible derived type including the type being defined.

C440 (R440) If the POINTER or ALLOCATABLE attribute is specified, each *component-array-spec* shall be a *deferred-shape-spec-list*.

C441 (R440) If neither the POINTER attribute nor the ALLOCATABLE attribute is specified, each *component-array-spec* shall be an *explicit-shape-spec-list*.

C442 (R443) Each bound in the *explicit-shape-spec* shall either be an initialization expression or be a specification expression that does not contain references to specification functions or any object designators other than named constants or subobjects thereof.

C443 (R440) A component shall not have both the ALLOCATABLE and the POINTER attribute.

C444 (R442) The * *char-length* option is permitted only if the type specified is character.

C445 (R439) Each *type-param-value* within a *component-def-stmt* shall either be a colon, be an initialization expression, or be a specification expression that contains neither references to specification functions nor any object designators other than named constants or subobjects thereof.

C446 (R440) If *component-initialization* appears, a double-colon separator shall appear before the *component-decl-list*.

C447 (R440) If => appears in *component-initialization*, POINTER shall appear in the *component-attr-spec-list*. If = appears in *component-initialization*, POINTER or ALLOCATABLE shall not appear in the *component-attr-spec-list*.

R445 *proc-component-def-stmt* **is** PROCEDURE ([*proc-interface*]) , ■
■ *proc-component-attr-spec-list* :: *proc-decl-list*

R446 *proc-component-attr-spec* **is** POINTER
or PASS [(*arg-name*)]
or NOPASS
or *access-spec*

C448 (R445) The same *proc-component-attr-spec* shall not appear more than once in a given *proc-component-def-stmt*.

C449 (R445) POINTER shall appear in each *proc-component-attr-spec-list*.

C450 (R445) If the procedure pointer component has an implicit interface or has no arguments, NOPASS shall be specified.

C451 (R445) If PASS (*arg-name*) appears, the interface shall have a dummy argument named *arg-name*.

C452 (R445) PASS and NOPASS shall not both appear in the same *proc-component-attr-spec-list*.

C453 The passed-object dummy argument shall be a scalar, nonpointer, nonallocatable dummy data object with the same declared type as the type being defined; all of its length type parameters shall be assumed; it shall be polymorphic (5.1.1.2) if and only if the type being defined is extensible (4.5.6).

R447 *private-components-stmt* **is** PRIVATE

C454 (R447) A *private-components-stmt* is permitted only if the type definition is within the specification part of a module.

R448 *type-bound-procedure-part* **is** *contains-stmt*
[*binding-private-stmt*]
proc-binding-stmt
[*proc-binding-stmt*] ...

R449 *binding-private-stmt* **is** PRIVATE

C455 (R448) A *binding-private-stmt* is permitted only if the type definition is within the specification part of a module.

R450 *proc-binding-stmt* **is** *specific-binding*
or *generic-binding*
or *final-binding*

R451 *specific-binding* **is** PROCEDURE [(*interface-name*)] ■
■ [[, *binding-attr-list*] ::] ■
■ *binding-name* [=> *procedure-name*]

C456 (R451) If => *procedure-name* appears, the double-colon separator shall appear.

C457 (R451) If => *procedure-name* appears, *interface-name* shall not appear.

C458 (R451) The *procedure-name* shall be the name of an accessible module procedure or an external procedure that has an explicit interface.

R452 *generic-binding* **is** GENERIC ■
■ [, *access-spec*] :: *generic-spec* => *binding-name-list*

C459 (R452) Within the *specification-part* of a module, each *generic-binding* shall specify, either implicitly or explicitly, the same accessibility as every other *generic-binding* with that *generic-spec* in the same derived type.

C460 (R452) Each *binding-name* in *binding-name-list* shall be the name of a specific binding of the type.

C461 (R452) If *generic-spec* is not *generic-name*, each of its specific bindings shall have a passed-object dummy argument (4.5.3.3).

C462 (R452) If *generic-spec* is OPERATOR (*defined-operator*), the interface of each binding shall be as specified in 12.3.2.1.1.

C463 (R452) If *generic-spec* is ASSIGNMENT (=), the interface of each binding shall be as specified in 12.3.2.1.2.

C464 (R452) If *generic-spec* is *dtio-generic-spec*, the interface of each binding shall be as specified in 9.5.3.7. The type of the `dtv` argument shall be *type-name*.

R453 *binding-attr* **is** PASS [(*arg-name*)]
or NOPASS
or NON_OVERRIDABLE
or DEFERRED
or *access-spec*

C465 (R453) The same *binding-attr* shall not appear more than once in a given *binding-attr-list*.

C466 (R451) If the interface of the binding has no dummy argument of the type being defined, NOPASS shall appear.

C467 (R451) If PASS (*arg-name*) appears, the interface of the binding shall have a dummy argument named *arg-name*.

C468 (R453) PASS and NOPASS shall not both appear in the same *binding-attr-list*.

C469 (R453) NON_OVERRIDABLE and DEFERRED shall not both appear in the same *binding-attr-list*.

C470 (R453) DEFERRED shall appear if and only if *interface-name* appears.

C471 (R451) An overriding binding (4.5.6.2) shall have the DEFERRED attribute only if the binding it overrides is deferred.

C472 (R451) A binding shall not override an inherited binding (4.5.6.1) that has the NON_OVERRIDABLE attribute.

R454 *final-binding* **is** FINAL [::] *final-subroutine-name-list*

C473 (R454) A *final-subroutine-name* shall be the name of a module procedure with exactly one dummy argument. That argument shall be nonoptional and shall be a nonpointer, nonallocatable, nonpolymorphic variable of the derived type being defined. All length type parameters of the dummy argument shall be assumed. The dummy argument shall not be INTENT(OUT).

C474 (R454) A *final-subroutine-name* shall not be one previously specified as a final subroutine for that type.

C475 (R454) A final subroutine shall not have a dummy argument with the same kind type parameters and rank as the dummy argument of another final subroutine of that type.

R455 *derived-type-spec* **is** *type-name* [(*type-param-spec-list*)]

R456 *type-param-spec* **is** [*keyword* =] *type-param-value*

C476 (R455) *type-name* shall be the name of an accessible derived type.

C477 (R455) *type-param-spec-list* shall appear only if the type is parameterized.

C478 (R455) There shall be at most one *type-param-spec* corresponding to each parameter of the type. If a type parameter does not have a default value, there shall be a *type-param-spec* corresponding to that type parameter.

C479 (R456) The *keyword*= may be omitted from a *type-param-spec* only if the *keyword*= has been omitted from each preceding *type-param-spec* in the *type-param-spec-list*.

C480 (R456) Each *keyword* shall be the name of a parameter of the type.

C481 (R456) An asterisk may be used as a *type-param-value* in a *type-param-spec* only in the declaration of a dummy argument or associate name or in the allocation of a dummy argument.

R457	*structure-constructor*	**is**	*derived-type-spec* ([*component-spec-list*])
R458	*component-spec*	**is**	[*keyword* =] *component-data-source*
R459	*component-data-source*	**is**	*expr*
		or	*data-target*
		or	*proc-target*

C482 (R457) The *derived-type-spec* shall not specify an abstract type (4.5.6).

C483 (R457) At most one *component-spec* shall be provided for a component.

C484 (R457) If a *component-spec* is provided for a component, no *component-spec* shall be provided for any component with which it is inheritance associated.

C485 (R457) A *component-spec* shall be provided for a component unless it has default initialization or is inheritance associated with another component for which a *component-spec* is provided or that has default initialization.

C486 (R458) The *keyword*= may be omitted from a *component-spec* only if the *keyword*= has been omitted from each preceding *component-spec* in the constructor.

C487 (R458) Each *keyword* shall be the name of a component of the type.

C488 (R457) The type name and all components of the type for which a *component-spec* appears shall be accessible in the scoping unit containing the structure constructor.

C489 (R457) If *derived-type-spec* is a type name that is the same as a generic name, the *component-spec-list* shall not be a valid *actual-arg-spec-list* for a function reference that is resolvable as a generic reference (12.4.4.1).

C490 (R459) A *data-target* shall correspond to a nonprocedure pointer component; a *proc-target* shall correspond to a procedure pointer component.

C491 (R459) A *data-target* shall have the same rank as its corresponding component.

R460	*enum-def*	**is**	*enum-def-stmt*
			enumerator-def-stmt
			[*enumerator-def-stmt*] ...
			end-enum-stmt
R461	*enum-def-stmt*	**is**	ENUM, BIND(C)
R462	*enumerator-def-stmt*	**is**	ENUMERATOR [::] *enumerator-list*
R463	*enumerator*	**is**	*named-constant* [= *scalar-int-initialization-expr*]
R464	*end-enum-stmt*	**is**	END ENUM

C492 (R462) If = appears in an *enumerator*, a double-colon separator shall appear before the *enumerator-list*.

R465	*array-constructor*	**is**	(/ *ac-spec* /)
		or	*left-square-bracket ac-spec right-square-bracket*
R466	*ac-spec*	**is**	*type-spec* ::
		or	[*type-spec* ::] *ac-value-list*
R467	*left-square-bracket*	**is**	[
R468	*right-square-bracket*	**is**	]
R469	*ac-value*	**is**	*expr*
		or	*ac-implied-do*
R470	*ac-implied-do*	**is**	(*ac-value-list* , *ac-implied-do-control*)

R471 *ac-implied-do-control* **is** *ac-do-variable* = *scalar-int-expr* , *scalar-int-expr* ■
■ [, *scalar-int-expr*]

R472 *ac-do-variable* **is** *scalar-int-variable*

C493 (R472) *ac-do-variable* shall be a named variable.

C494 (R466) If *type-spec* is omitted, each *ac-value* expression in the *array-constructor* shall have the same type and kind type parameters.

C495 (R466) If *type-spec* specifies an intrinsic type, each *ac-value* expression in the *array-constructor* shall be of an intrinsic type that is in type conformance with a variable of type *type-spec* as specified in Table 7.8.

C496 (R466) If *type-spec* specifies a derived type, all *ac-value* expressions in the *array-constructor* shall be of that derived type and shall have the same kind type parameter values as specified by *type-spec*.

C497 (R470) The *ac-do-variable* of an *ac-implied-do* that is in another *ac-implied-do* shall not appear as the *ac-do-variable* of the containing *ac-implied-do*.

Section 5:

R501 *type-declaration-stmt* **is** *declaration-type-spec* [[, *attr-spec*] ... ::] *entity-decl-list*

R502 *declaration-type-spec* **is** *intrinsic-type-spec*
or TYPE (*derived-type-spec*)
or CLASS (*derived-type-spec*)
or CLASS (*)

C501 (R502) In a *declaration-type-spec*, every *type-param-value* that is not a colon or an asterisk shall be a *specification-expr*.

C502 (R502) In a *declaration-type-spec* that uses the CLASS keyword, *derived-type-spec* shall specify an extensible type.

C503 (R502) The TYPE(*derived-type-spec*) shall not specify an abstract type (4.5.6).

R503 *attr-spec* **is** *access-spec*
or ALLOCATABLE
or ASYNCHRONOUS
or DIMENSION (*array-spec*)
or EXTERNAL
or INTENT (*intent-spec*)
or INTRINSIC
or *language-binding-spec*
or OPTIONAL
or PARAMETER
or POINTER
or PROTECTED
or SAVE
or TARGET
or VALUE
or VOLATILE

R504 *entity-decl* **is** *object-name* [(*array-spec*)] [* *char-length*] [*initialization*]
or *function-name* [* *char-length*]

C504 (R504) If a *type-param-value* in a *char-length* in an *entity-decl* is not a colon or an asterisk, it shall be a *specification-expr*.

R505 *object-name* **is** *name*

C505 (R505) The *object-name* shall be the name of a data object.

R506	*initialization*	**is**	= *initialization-expr*
		or	=> *null-init*
R507	*null-init*	**is**	*function-reference*

C506 (R507) The *function-reference* shall be a reference to the NULL intrinsic function with no arguments.

C507 (R501) The same *attr-spec* shall not appear more than once in a given *type-declaration-stmt*.

C508 An entity shall not be explicitly given any attribute more than once in a scoping unit.

C509 (R501) An entity declared with the CLASS keyword shall be a dummy argument or have the ALLOCATABLE or POINTER attribute.

C510 (R501) An array that has the POINTER or ALLOCATABLE attribute shall be specified with an *array-spec* that is a *deferred-shape-spec-list* (5.1.2.5.3).

C511 (R501) An *array-spec* for an *object-name* that is a function result that does not have the ALLOCATABLE or POINTER attribute shall be an *explicit-shape-spec-list*.

C512 (R501) If the POINTER attribute is specified, the ALLOCATABLE, TARGET, EXTERNAL, or INTRINSIC attribute shall not be specified.

C513 (R501) If the TARGET attribute is specified, the POINTER, EXTERNAL, INTRINSIC, or PARAMETER attribute shall not be specified.

C514 (R501) The PARAMETER attribute shall not be specified for a dummy argument, a pointer, an allocatable entity, a function, or an object in a common block.

C515 (R501) The INTENT, VALUE, and OPTIONAL attributes may be specified only for dummy arguments.

C516 (R501) The INTENT attribute shall not be specified for a dummy procedure without the POINTER attribute.

C517 (R501) The SAVE attribute shall not be specified for an object that is in a common block, a dummy argument, a procedure, a function result, an automatic data object, or an object with the PARAMETER attribute.

C518 An entity shall not have both the EXTERNAL attribute and the INTRINSIC attribute.

C519 (R501) An entity in an *entity-decl-list* shall not have the EXTERNAL or INTRINSIC attribute specified unless it is a function.

C520 (R504) The * *char-length* option is permitted only if the type specified is character.

C521 (R504) The *function-name* shall be the name of an external function, an intrinsic function, a function dummy procedure, or a statement function.

C522 (R501) The *initialization* shall appear if the statement contains a PARAMETER attribute (5.1.2.10).

C523 (R501) If *initialization* appears, a double-colon separator shall appear before the *entity-decl-list*.

C524 (R504)*initialization* shall not appear if *object-name* is a dummy argument, a function result, an object in a named common block unless the type declaration is in a block data program unit, an object in blank common, an allocatable variable, an external name, an intrinsic name, or an automatic object.

C525 (R504) If => appears in *initialization*, the object shall have the POINTER attribute. If = appears in *initialization*, the object shall not have the POINTER attribute.

C526 (R501) If the VOLATILE attribute is specified, the PARAMETER, INTRINSIC, EXTERNAL, or INTENT(IN) attribute shall not be specified.

C527 (R501) If the VALUE attribute is specified, the PARAMETER, EXTERNAL, POINTER, ALLOCATABLE, DIMENSION, VOLATILE, INTENT(INOUT), or INTENT(OUT) attribute shall not be specified.

C528 (R501) If the VALUE attribute is specified for a dummy argument of type character, the length parameter shall be omitted or shall be specified by an initialization expression with the value one.

C529 (R501) The VALUE attribute shall not be specified for a dummy procedure.

C530 (R501) The ALLOCATABLE, POINTER, or OPTIONAL attribute shall not be specified for a dummy argument of a procedure that has a *proc-language-binding-spec*.

C531 (R503) A *language-binding-spec* shall appear only in the specification part of a module.

C532 (R501) If a *language-binding-spec* is specified, the entity declared shall be an interoperable variable (15.2).

C533 (R501) If a *language-binding-spec* with a NAME= specifier appears, the *entity-decl-list* shall consist of a single *entity-decl*.

C534 (R503) The PROTECTED attribute is permitted only in the specification part of a module.

C535 (R501) The PROTECTED attribute is permitted only for a procedure pointer or named variable that is not in a common block.

C536 (R501) If the PROTECTED attribute is specified, the EXTERNAL, INTRINSIC, or PARAMETER attribute shall not be specified.

C537 A nonpointer object that has the PROTECTED attribute and is accessed by use association shall not appear in a variable definition context (16.5.7) or as the *data-target* or *proc-target* in a *pointer-assignment-stmt*.

C538 A pointer object that has the PROTECTED attribute and is accessed by use association shall not appear as

(1) A *pointer-object* in a *pointer-assignment-stmt* or *nullify-stmt*,

(2) An *allocate-object* in an *allocate-stmt* or *deallocate-stmt*, or

(3) An actual argument in a reference to a procedure if the associated dummy argument is a pointer with the INTENT(OUT) or INTENT(INOUT) attribute.

R508	*access-spec*	**is**	PUBLIC
		or	PRIVATE

C539 (R508) An *access-spec* shall appear only in the *specification-part* of a module.

R509	*language-binding-spec*	**is**	BIND (C [, NAME = *scalar-char-initialization-expr*])

C540 (R509) The *scalar-char-initialization-expr* shall be of default character kind.

R510	*array-spec*	**is**	*explicit-shape-spec-list*
		or	*assumed-shape-spec-list*
		or	*deferred-shape-spec-list*
		or	*assumed-size-spec*

C541 (R510)The maximum rank is seven.

R511	*explicit-shape-spec*	**is**	[*lower-bound* :] *upper-bound*
R512	*lower-bound*	**is**	*specification-expr*
R513	*upper-bound*	**is**	*specification-expr*

C542 (R511) An explicit-shape array whose bounds are not initialization expressions shall be a dummy argument, a function result, or an automatic array of a procedure.

R514	*assumed-shape-spec*	**is**	[*lower-bound*] :
R515	*deferred-shape-spec*	**is**	:
R516	*assumed-size-spec*	**is**	[*explicit-shape-spec-list* ,] [*lower-bound* :] *

C543 An *assumed-size-spec* shall not appear except as the declaration of the array bounds of a dummy data argument.

C544 An assumed-size array with INTENT (OUT) shall not be of a type for which default initialization is specified.

R517	*intent-spec*	**is**	IN
		or	OUT

or INOUT

C545 (R517) A nonpointer object with the INTENT (IN) attribute shall not appear in a variable definition context (16.5.7).

C546 (R517) A pointer object with the INTENT (IN) attribute shall not appear as

C547 (R503) (R1216) If the name of a generic intrinsic procedure is explicitly declared to have the INTRINSIC attribute, and it is also the generic name in one or more generic interfaces (12.3.2.1) accessible in the same scoping unit, the procedures in the interfaces and the specific intrinsic procedures shall all be functions or all be subroutines, and the characteristics of the specific intrinsic procedures and the procedures in the interfaces shall differ as specified in 16.2.3.

R518 *access-stmt* **is** *access-spec* [[::] *access-id-list*]

R519 *access-id* **is** *use-name*
or *generic-spec*

C548 (R518) An *access-stmt* shall appear only in the *specification-part* of a module. Only one accessibility statement with an omitted *access-id-list* is permitted in the *specification-part* of a module.

C549 (R519) Each *use-name* shall be the name of a named variable, procedure, derived type, named constant, or namelist group.

R520 *allocatable-stmt* **is** ALLOCATABLE [::] ■
■ *object-name* [(*deferred-shape-spec-list*)] ■
■ [, *object-name* [(*deferred-shape-spec-list*)]] ...

R521 *asynchronous-stmt* **is** ASYNCHRONOUS [::] *object-name-list*

R522 *bind-stmt* **is** *language-binding-spec* [::] *bind-entity-list*

R523 *bind-entity* **is** *entity-name*
or / *common-block-name* /

C550 (R522) If any *bind-entity* in a *bind-stmt* is an *entity-name*, the *bind-stmt* shall appear in the specification part of a module and the entity shall be an interoperable variable (15.2.4, 15.2.5).

C551 (R522) If the *language-binding-spec* has a NAME= specifier, the *bind-entity-list* shall consist of a single *bind-entity*.

C552 (R522) If a *bind-entity* is a common block, each variable of the common block shall be interoperable (15.2.4, 15.2.5).

R524 *data-stmt* **is** DATA *data-stmt-set* [[,] *data-stmt-set*] ...

R525 *data-stmt-set* **is** *data-stmt-object-list* / *data-stmt-value-list* /

R526 *data-stmt-object* **is** *variable*
or *data-implied-do*

R527 *data-implied-do* **is** (*data-i-do-object-list* , *data-i-do-variable* = ■
■ *scalar-int-expr* , *scalar-int-expr* [, *scalar-int-expr*])

R528 *data-i-do-object* **is** *array-element*
or *scalar-structure-component*
or *data-implied-do*

R529 *data-i-do-variable* **is** *scalar-int-variable*

C553 (R526) In a *variable* that is a *data-stmt-object*, any subscript, section subscript, substring starting point, and substring ending point shall be an initialization expression.

C554 (R526) A variable whose designator is included in a *data-stmt-object-list* or a *data-i-do-object-list* shall not be: a dummy argument, made accessible by use association or host association, in a named common block unless the DATA statement is in a block data program unit, in a blank common block, a function name, a function result name, an automatic object, or an allocatable variable.

C555 (R526) A *data-i-do-object* or a *variable* that appears as a *data-stmt-object* shall not be an object designator in which a pointer appears other than as the entire rightmost *part-ref*.

C556 (R529) The *data-i-do-variable* shall be a named variable.

C557 (R527) A *scalar-int-expr* of a *data-implied-do* shall involve as primaries only constants, subobjects of constants, or DO variables of the containing *data-implied-do*s, and each operation shall be intrinsic.

C558 (R528) The *array-element* shall be a variable.

C559 (R528) The *scalar-structure-component* shall be a variable.

C560 (R528) The *scalar-structure-component* shall contain at least one *part-ref* that contains a *subscript-list*.

C561 (R528) In an *array-element* or a *scalar-structure-component* that is a *data-i-do-object*, any subscript shall be an expression whose primaries are either constants, subobjects of constants, or DO variables of this *data-implied-do* or the containing *data-implied-do*s, and each operation shall be intrinsic.

R530	*data-stmt-value*	**is**	[*data-stmt-repeat* *] *data-stmt-constant*
R531	*data-stmt-repeat*	**is**	*scalar-int-constant*
		or	*scalar-int-constant-subobject*

C562 (R531) The *data-stmt-repeat* shall be positive or zero. If the *data-stmt-repeat* is a named constant, it shall have been declared previously in the scoping unit or made accessible by use association or host association.

R532	*data-stmt-constant*	**is**	*scalar-constant*
		or	*scalar-constant-subobject*
		or	*signed-int-literal-constant*
		or	*signed-real-literal-constant*
		or	*null-init*
		or	*structure-constructor*

C563 (R532) If a DATA statement constant value is a named constant or a structure constructor, the named constant or derived type shall have been declared previously in the scoping unit or made accessible by use or host association.

C564 (R532) If a *data-stmt-constant* is a *structure-constructor*, it shall be an initialization expression.

R533	*int-constant-subobject*	**is**	*constant-subobject*

C565 (R533) *int-constant-subobject* shall be of type integer.

R534	*constant-subobject*	**is**	*designator*

C566 (R534) *constant-subobject* shall be a subobject of a constant.

C567 (R534) Any subscript, substring starting point, or substring ending point shall be an initialization expression.

R535	*dimension-stmt*	**is**	DIMENSION [::] *array-name* (*array-spec*) ■
			■ [, *array-name* (*array-spec*)] ...
R536	*intent-stmt*	**is**	INTENT (*intent-spec*) [::] *dummy-arg-name-list*
R537	*optional-stmt*	**is**	OPTIONAL [::] *dummy-arg-name-list*
R538	*parameter-stmt*	**is**	PARAMETER (*named-constant-def-list*)
R539	*named-constant-def*	**is**	*named-constant* = *initialization-expr*
R540	*pointer-stmt*	**is**	POINTER [::] *pointer-decl-list*
R541	*pointer-decl*	**is**	*object-name* [(*deferred-shape-spec-list*)]
		or	*proc-entity-name*

C568 (R541) A *proc-entity-name* shall also be declared in a *procedure-declaration-stmt*.

R542	*protected-stmt*	**is**	PROTECTED [::] *entity-name-list*
R543	*save-stmt*	**is**	SAVE [[::] *saved-entity-list*]
R544	*saved-entity*	**is**	*object-name*

		or	*proc-pointer-name*
		or	/ *common-block-name* /
R545	*proc-pointer-name*	**is**	*name*

C569 (R545) A *proc-pointer-name* shall be the name of a procedure pointer.

C570 (R543) If a SAVE statement with an omitted saved entity list occurs in a scoping unit, no other explicit occurrence of the SAVE attribute or SAVE statement is permitted in the same scoping unit.

R546	*target-stmt*	**is**	TARGET [::] *object-name* [(*array-spec*)] ■
			■ [, *object-name* [(*array-spec*)]] ...
R547	*value-stmt*	**is**	VALUE [::] *dummy-arg-name-list*
R548	*volatile-stmt*	**is**	VOLATILE [::] *object-name-list*
R549	*implicit-stmt*	**is**	IMPLICIT *implicit-spec-list*
		or	IMPLICIT NONE
R550	*implicit-spec*	**is**	*declaration-type-spec* (*letter-spec-list*)
R551	*letter-spec*	**is**	*letter* [– *letter*]

C571 (R549) If IMPLICIT NONE is specified in a scoping unit, it shall precede any PARAMETER statements that appear in the scoping unit and there shall be no other IMPLICIT statements in the scoping unit.

C572 (R551) If the minus and second letter appear, the second letter shall follow the first letter alphabetically.

R552	*namelist-stmt*	**is**	NAMELIST ■
			■ / *namelist-group-name* / *namelist-group-object-list* ■
			■ [[,] / *namelist-group-name* / ■
			■ *namelist-group-object-list*] . . .

C573 (R552) The *namelist-group-name* shall not be a name made accessible by use association.

R553	*namelist-group-object*	**is**	*variable-name*

C574 (R553) A *namelist-group-object* shall not be an assumed-size array.

C575 (R552) A *namelist-group-object* shall not have the PRIVATE attribute if the *namelist-group-name* has the PUBLIC attribute.

R554	*equivalence-stmt*	**is**	EQUIVALENCE *equivalence-set-list*
R555	*equivalence-set*	**is**	(*equivalence-object* , *equivalence-object-list*)
R556	*equivalence-object*	**is**	*variable-name*
		or	*array-element*
		or	*substring*

C576 (R556) An *equivalence-object* shall not be a designator with a base object that is a dummy argument, a pointer, an allocatable variable, a derived-type object that has an allocatable ultimate component, an object of a nonsequence derived type, an object of a derived type that has a pointer at any level of component selection, an automatic object, a function name, an entry name, a result name, a variable with the BIND attribute, a variable in a common block that has the BIND attribute, or a named constant.

C577 (R556) An *equivalence-object* shall not be a designator that has more than one *part-ref* .

C578 (R556) An *equivalence-object* shall not have the TARGET attribute.

C579 (R556) Each subscript or substring range expression in an *equivalence-object* shall be an integer initialization expression (7.1.7).

C580 (R555) If an *equivalence-object* is of type default integer, default real, double precision real, default complex, default logical, or numeric sequence type, all of the objects in the equivalence set shall be of these types.

C581 (R555) If an *equivalence-object* is of type default character or character sequence type, all of the objects in the equivalence set shall be of these types.

C582 (R555) If an *equivalence-object* is of a sequence derived type that is not a numeric sequence or character sequence type, all of the objects in the equivalence set shall be of the same type with the same type parameter values.

C583 (R555) If an *equivalence-object* is of an intrinsic type other than default integer, default real, double precision real, default complex, default logical, or default character, all of the objects in the equivalence set shall be of the same type with the same kind type parameter value.

C584 (R556) If an *equivalence-object* has the PROTECTED attribute, all of the objects in the equivalence set shall have the PROTECTED attribute.

C585 (R556) The name of an *equivalence-object* shall not be a name made accessible by use association.

C586 (R556) A *substring* shall not have length zero.

R557	*common-stmt*	**is**	COMMON ■
			■ [/ [*common-block-name*] /] *common-block-object-list* ■
			■ [[,] / [*common-block-name*] / ■
			■ *common-block-object-list*] ...
R558	*common-block-object*	**is**	*variable-name* [(*explicit-shape-spec-list*)]
		or	*proc-pointer-name*

C587 (R558) Only one appearance of a given *variable-name* or *proc-pointer-name* is permitted in all *common-block-object-list*s within a scoping unit.

C588 (R558) A *common-block-object* shall not be a dummy argument, an allocatable variable, a derived-type object with an ultimate component that is allocatable, an automatic object, a function name, an entry name, a variable with the BIND attribute, or a result name.

C589 (R558) If a *common-block-object* is of a derived type, it shall be a sequence type (4.5.1) or a type with the BIND attribute and it shall have no default initialization.

C590 (R558) A *variable-name* or *proc-pointer-name* shall not be a name made accessible by use association.

Section 6:

R601 *variable* **is** *designator*

C601 (R601) *designator* shall not be a constant or a subobject of a constant.

R602 *variable-name* **is** *name*

C602 (R602) A *variable-name* shall be the name of a variable.

R603	*designator*	**is**	*object-name*
		or	*array-element*
		or	*array-section*
		or	*structure-component*
		or	*substring*

R604 *logical-variable* **is** *variable*

C603 (R604) *logical-variable* shall be of type logical.

R605 *default-logical-variable* **is** *variable*

C604 (R605) *default-logical-variable* shall be of type default logical.

R606 *char-variable* **is** *variable*

C605 (R606) *char-variable* shall be of type character.

R607 *default-char-variable* **is** *variable*

C606 (R607) *default-char-variable* shall be of type default character.

R608 *int-variable* **is** *variable*

C607 (R608) *int-variable* shall be of type integer.

R609 *substring* **is** *parent-string* (*substring-range*)

R610	*parent-string*	**is**	*scalar-variable-name*
		or	*array-element*
		or	*scalar-structure-component*
		or	*scalar-constant*
R611	*substring-range*	**is**	[*scalar-int-expr*] : [*scalar-int-expr*]

C608 (R610) *parent-string* shall be of type character.

R612	*data-ref*	**is**	*part-ref* [% *part-ref*] ...
R613	*part-ref*	**is**	*part-name* [(*section-subscript-list*)]

C609 (R612) Each *part-name* except the rightmost shall be of derived type.

C610 (R612) Each *part-name* except the leftmost shall be the name of a component of the declared type of the preceding *part-name*.

C611 (R612) If the rightmost *part-name* is of abstract type, *data-ref* shall be polymorphic.

C612 (R612) The leftmost *part-name* shall be the name of a data object.

C613 (R613) If a *section-subscript-list* appears, the number of *section-subscripts* shall equal the rank of *part-name*.

C614 (R612) There shall not be more than one *part-ref* with nonzero rank. A *part-name* to the right of a *part-ref* with nonzero rank shall not have the ALLOCATABLE or POINTER attribute.

R614	*structure-component*	**is**	*data-ref*

C615 (R614) There shall be more than one *part-ref* and the rightmost *part-ref* shall be of the form *part-name*.

R615	*type-param-inquiry*	**is**	*designator* % *type-param-name*

C616 (R615) The *type-param-name* shall be the name of a type parameter of the declared type of the object designated by the *designator*.

R616	*array-element*	**is**	*data-ref*

C617 (R616) Every *part-ref* shall have rank zero and the last *part-ref* shall contain a *subscript-list*.

R617	*array-section*	**is**	*data-ref* [(*substring-range*)]

C618 (R617) Exactly one *part-ref* shall have nonzero rank, and either the final *part-ref* shall have a *section-subscript-list* with nonzero rank or another *part-ref* shall have nonzero rank.

C619 (R617) If a *substring-range* appears, the rightmost *part-name* shall be of type character.

R618	*subscript*	**is**	*scalar-int-expr*
R619	*section-subscript*	**is**	*subscript*
		or	*subscript-triplet*
		or	*vector-subscript*
R620	*subscript-triplet*	**is**	[*subscript*] : [*subscript*] [: *stride*]
R621	*stride*	**is**	*scalar-int-expr*
R622	*vector-subscript*	**is**	*int-expr*

C620 (R622) A *vector-subscript* shall be an integer array expression of rank one.

C621 (R620) The second subscript shall not be omitted from a *subscript-triplet* in the last dimension of an assumed-size array.

R623	*allocate-stmt*	**is**	ALLOCATE ([*type-spec* ::] *allocation-list* ■
			■ [, *alloc-opt-list*])
R624	*alloc-opt*	**is**	STAT = *stat-variable*
		or	ERRMSG = *errmsg-variable*
		or	SOURCE = *source-expr*
R625	*stat-variable*	**is**	*scalar-int-variable*
R626	*errmsg-variable*	**is**	*scalar-default-char-variable*
R627	*source-expr*	**is**	*expr*

R628	*allocation*	**is**	*allocate-object* [(*allocate-shape-spec-list*)]
R629	*allocate-object*	**is**	*variable-name*
		or	*structure-component*
R630	*allocate-shape-spec*	**is**	[*lower-bound-expr* :] *upper-bound-expr*
R631	*lower-bound-expr*	**is**	*scalar-int-expr*
R632	*upper-bound-expr*	**is**	*scalar-int-expr*

C622 (R629) Each *allocate-object* shall be a nonprocedure pointer or an allocatable variable.

C623 (R623) If any *allocate-object* in the statement has a deferred type parameter, either *type-spec* or SOURCE= shall appear.

C624 (R623) If a *type-spec* appears, it shall specify a type with which each *allocate-object* is type compatible.

C625 (R623) If any *allocate-object* is unlimited polymorphic, either *type-spec* or SOURCE= shall appear.

C626 (R623) A *type-param-value* in a *type-spec* shall be an asterisk if and only if each *allocate-object* is a dummy argument for which the corresponding type parameter is assumed.

C627 (R623) If a *type-spec* appears, the kind type parameter values of each *allocate-object* shall be the same as the corresponding type parameter values of the *type-spec*.

C628 (R628) An *allocate-shape-spec-list* shall appear if and only if the *allocate-object* is an array.

C629 (R628) The number of *allocate-shape-spec*s in an *allocate-shape-spec-list* shall be the same as the rank of the *allocate-object*.

C630 (R624) No *alloc-opt* shall appear more than once in a given *alloc-opt-list*.

C631 (R623) If SOURCE= appears, *type-spec* shall not appear and *allocation-list* shall contain only one *allocate-object*, which shall be type compatible (5.1.1.2) with *source-expr*.

C632 (R623) The *source-expr* shall be a scalar or have the same rank as *allocate-object*.

C633 (R623) Corresponding kind type parameters of *allocate-object* and *source-expr* shall have the same values.

R633	*nullify-stmt*	**is**	NULLIFY (*pointer-object-list*)
R634	*pointer-object*	**is**	*variable-name*
		or	*structure-component*
		or	*proc-pointer-name*

C634 (R634) Each *pointer-object* shall have the POINTER attribute.

R635	*deallocate-stmt*	**is**	DEALLOCATE (*allocate-object-list* [, *dealloc-opt-list*])

C635 (R635) Each *allocate-object* shall be a nonprocedure pointer or an allocatable variable.

R636	*dealloc-opt*	**is**	STAT = *stat-variable*
		or	ERRMSG = *errmsg-variable*

C636 (R636) No *dealloc-opt* shall appear more than once in a given *dealloc-opt-list*.

Section 7:

R701	*primary*	**is**	*constant*
		or	*designator*
		or	*array-constructor*
		or	*structure-constructor*
		or	*function-reference*
		or	*type-param-inquiry*
		or	*type-param-name*
		or	(*expr*)

C701 (R701) The *type-param-name* shall be the name of a type parameter.

C702 (R701) The *designator* shall not be a whole assumed-size array.

R702	*level-1-expr*	**is**	[*defined-unary-op*] *primary*
R703	*defined-unary-op*	**is**	. *letter* [*letter*] ... .
C703	(R703) A *defined-unary-op* shall not contain more than 63 letters and shall not be the same as any *intrinsic-operator* or *logical-literal-constant*.		
R704	*mult-operand*	**is**	*level-1-expr* [*power-op mult-operand*]
R705	*add-operand*	**is**	[*add-operand mult-op*] *mult-operand*
R706	*level-2-expr*	**is**	[[*level-2-expr*] *add-op*] *add-operand*
R707	*power-op*	**is**	**
R708	*mult-op*	**is**	*
		or	/
R709	*add-op*	**is**	+
		or	−
R710	*level-3-expr*	**is**	[*level-3-expr concat-op*] *level-2-expr*
R711	*concat-op*	**is**	//
R712	*level-4-expr*	**is**	[*level-3-expr rel-op*] *level-3-expr*
R713	*rel-op*	**is**	.EQ.
		or	.NE.
		or	.LT.
		or	.LE.
		or	.GT.
		or	.GE.
		or	==
		or	/=
		or	<
		or	<=
		or	>
		or	>=
R714	*and-operand*	**is**	[*not-op*] *level-4-expr*
R715	*or-operand*	**is**	[*or-operand and-op*] *and-operand*
R716	*equiv-operand*	**is**	[*equiv-operand or-op*] *or-operand*
R717	*level-5-expr*	**is**	[*level-5-expr equiv-op*] *equiv-operand*
R718	*not-op*	**is**	.NOT.
R719	*and-op*	**is**	.AND.
R720	*or-op*	**is**	.OR.
R721	*equiv-op*	**is**	.EQV.
		or	.NEQV.
R722	*expr*	**is**	[*expr defined-binary-op*] *level-5-expr*
R723	*defined-binary-op*	**is**	. *letter* [*letter*] ... .
C704	(R723) A *defined-binary-op* shall not contain more than 63 letters and shall not be the same as any *intrinsic-operator* or *logical-literal-constant*.		
R724	*logical-expr*	**is**	*expr*
C705	(R724) *logical-expr* shall be of type logical.		
R725	*char-expr*	**is**	*expr*
C706	(R725) *char-expr* shall be of type character.		
R726	*default-char-expr*	**is**	*expr*
C707	(R726) *default-char-expr* shall be of type default character.		

R727 *int-expr* **is** *expr*

C708 (R727) *int-expr* shall be of type integer.

R728 *numeric-expr* **is** *expr*

C709 (R728) *numeric-expr* shall be of type integer, real, or complex.

R729 *specification-expr* **is** *scalar-int-expr*

C710 (R729) The *scalar-int-expr* shall be a restricted expression.

R730 *initialization-expr* **is** *expr*

C711 (R730) *initialization-expr* shall be an initialization expression.

R731 *char-initialization-expr* **is** *char-expr*

C712 (R731) *char-initialization-expr* shall be an initialization expression.

R732 *int-initialization-expr* **is** *int-expr*

C713 (R732) *int-initialization-expr* shall be an initialization expression.

R733 *logical-initialization-expr* **is** *logical-expr*

C714 (R733) *logical-initialization-expr* shall be an initialization expression.

R734 *assignment-stmt* **is** *variable* = *expr*

C715 (R734) The *variable* in an *assignment-stmt* shall not be a whole assumed-size array.

R735 *pointer-assignment-stmt* **is** *data-pointer-object* [(*bounds-spec-list*)] => *data-target*
or *data-pointer-object* (*bounds-remapping-list*) => *data-target*
or *proc-pointer-object* => *proc-target*

R736 *data-pointer-object* **is** *variable-name*
or *variable* % *data-pointer-component-name*

C716 (R735) If *data-target* is not unlimited polymorphic, *data-pointer-object* shall be type compatible (5.1.1.2) with it, and the corresponding kind type parameters shall be equal.

C717 (R735) If *data-target* is unlimited polymorphic, *data-pointer-object* shall be unlimited polymorphic, of a sequence derived type, or of a type with the BIND attribute.

C718 (R735) If *bounds-spec-list* is specified, the number of *bounds-spec*s shall equal the rank of *data-pointer-object*.

C719 (R735) If *bounds-remapping-list* is specified, the number of *bounds-remapping*s shall equal the rank of *data-pointer-object*.

C720 (R735) If *bounds-remapping-list* is specified, *data-target* shall have rank one; otherwise, the ranks of *data-pointer-object* and *data-target* shall be the same.

C721 (R736) A *variable-name* shall have the POINTER attribute.

C722 (R736) A *data-pointer-component-name* shall be the name of a component of *variable* that is a data pointer.

R737 *bounds-spec* **is** *lower-bound-expr* :

R738 *bounds-remapping* **is** *lower-bound-expr* : *upper-bound-expr*

R739 *data-target* **is** *variable*
or *expr*

C723 (R739) A *variable* shall have either the TARGET or POINTER attribute, and shall not be an array section with a vector subscript.

C724 (R739) An *expr* shall be a reference to a function whose result is a data pointer.

R740 *proc-pointer-object* **is** *proc-pointer-name*
or *proc-component-ref*

R741 *proc-component-ref* **is** *variable* % *procedure-component-name*

C725 (R741) the *procedure-component-name* shall be the name of a procedure pointer component of the declared type of *variable*.

R742 *proc-target* **is** *expr*

		or	*procedure-name*
		or	*proc-component-ref*

C726 (R742) An *expr* shall be a reference to a function whose result is a procedure pointer.

C727 (R742) A *procedure-name* shall be the name of an external, module, or dummy procedure, a specific intrinsic function listed in 13.6 and not marked with a bullet (•), or a procedure pointer.

C728 (R742) The *proc-target* shall not be a nonintrinsic elemental procedure.

R743	*where-stmt*	**is**	WHERE (*mask-expr*) *where-assignment-stmt*
R744	*where-construct*	**is**	*where-construct-stmt* [*where-body-construct*] ... [*masked-elsewhere-stmt* [*where-body-construct*] ...] ... [*elsewhere-stmt* [*where-body-construct*] ...] *end-where-stmt*
R745	*where-construct-stmt*	**is**	[*where-construct-name*:] WHERE (*mask-expr*)
R746	*where-body-construct*	**is**	*where-assignment-stmt*
		or	*where-stmt*
		or	*where-construct*
R747	*where-assignment-stmt*	**is**	*assignment-stmt*
R748	*mask-expr*	**is**	*logical-expr*
R749	*masked-elsewhere-stmt*	**is**	ELSEWHERE (*mask-expr*) [*where-construct-name*]
R750	*elsewhere-stmt*	**is**	ELSEWHERE [*where-construct-name*]
R751	*end-where-stmt*	**is**	END WHERE [*where-construct-name*]

C729 (R747) A *where-assignment-stmt* that is a defined assignment shall be elemental.

C730 (R744) If the *where-construct-stmt* is identified by a *where-construct-name*, the corresponding *end-where-stmt* shall specify the same *where-construct-name*. If the *where-construct-stmt* is not identified by a *where-construct-name*, the corresponding *end-where-stmt* shall not specify a *where-construct-name*. If an *elsewhere-stmt* or a *masked-elsewhere-stmt* is identified by a *where-construct-name*, the corresponding *where-construct-stmt* shall specify the same *where-construct-name*.

C731 (R746) A statement that is part of a *where-body-construct* shall not be a branch target statement.

R752	*forall-construct*	**is**	*forall-construct-stmt* [*forall-body-construct*] ... *end-forall-stmt*
R753	*forall-construct-stmt*	**is**	[*forall-construct-name* :] FORALL *forall-header*
R754	*forall-header*	**is**	(*forall-triplet-spec-list* [, *scalar-mask-expr*])
R755	*forall-triplet-spec*	**is**	*index-name* = *subscript* : *subscript* [: *stride*]
R756	*forall-body-construct*	**is**	*forall-assignment-stmt*
		or	*where-stmt*
		or	*where-construct*
		or	*forall-construct*
		or	*forall-stmt*
R757	*forall-assignment-stmt*	**is**	*assignment-stmt*
		or	*pointer-assignment-stmt*
R758	*end-forall-stmt*	**is**	END FORALL [*forall-construct-name*]

C732 (R758) If the *forall-construct-stmt* has a *forall-construct-name*, the *end-forall-stmt* shall have the same *forall-construct-name*. If the *end-forall-stmt* has a *forall-construct-name*, the *forall-*

construct-stmt shall have the same *forall-construct-name*.

C733 (R754) The *scalar-mask-expr* shall be scalar and of type logical.

C734 (R754) Any procedure referenced in the *scalar-mask-expr*, including one referenced by a defined operation, shall be a pure procedure (12.6).

C735 (R755) The *index-name* shall be a named scalar variable of type integer.

C736 (R755) A *subscript* or *stride* in a *forall-triplet-spec* shall not contain a reference to any *index-name* in the *forall-triplet-spec-list* in which it appears.

C737 (R756) A statement in a *forall-body-construct* shall not define an *index-name* of the *forall-construct*.

C738 (R756) Any procedure referenced in a *forall-body-construct*, including one referenced by a defined operation, assignment, or finalization, shall be a pure procedure.

C739 (R756) A *forall-body-construct* shall not be a branch target.

R759 *forall-stmt* **is** FORALL *forall-header forall-assignment-stmt*

Section 8:

R801 *block* **is** [*execution-part-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-logical-expr*) THEN

R804 *else-if-stmt* **is** ELSE IF (*scalar-logical-expr*) THEN [*if-construct-name*]

R805 *else-stmt* **is** ELSE [*if-construct-name*]

R806 *end-if-stmt* **is** END IF [*if-construct-name*]

C801 (R802) If the *if-then-stmt* of an *if-construct* specifies an *if-construct-name*, the corresponding *end-if-stmt* shall specify the same *if-construct-name*. If the *if-then-stmt* of an *if-construct* does not specify an *if-construct-name*, the corresponding *end-if-stmt* shall not specify an *if-construct-name*. If an *else-if-stmt* or *else-stmt* specifies an *if-construct-name*, the corresponding *if-then-stmt* shall specify the same *if-construct-name*.

R807 *if-stmt* **is** IF (*scalar-logical-expr*) *action-stmt*

C802 (R807) The *action-stmt* in the *if-stmt* shall not be an *if-stmt*, *end-program-stmt*, *end-function-stmt*, or *end-subroutine-stmt*.

R808 *case-construct* **is** *select-case-stmt*
[*case-stmt*
block] ...
end-select-stmt

R809 *select-case-stmt* **is** [*case-construct-name* :] SELECT CASE (*case-expr*)

R810 *case-stmt* **is** CASE *case-selector* [*case-construct-name*]

R811 *end-select-stmt* **is** END SELECT [*case-construct-name*]

C803 (R808) If the *select-case-stmt* of a *case-construct* specifies a *case-construct-name*, the corresponding *end-select-stmt* shall specify the same *case-construct-name*. If the *select-case-stmt* of a *case-construct* does not specify a *case-construct-name*, the corresponding *end-select-stmt* shall not specify a *case-construct-name*. If a *case-stmt* specifies a *case-construct-name*, the corresponding *select-case-stmt* shall specify the same *case-construct-name*.

R812 *case-expr* **is** *scalar-int-expr*
or *scalar-char-expr*

		or	*scalar-logical-expr*
R813	*case-selector*	**is**	(*case-value-range-list*)
		or	DEFAULT

C804 (R808) No more than one of the selectors of one of the CASE statements shall be DEFAULT.

R814	*case-value-range*	**is**	*case-value*
		or	*case-value* :
		or	: *case-value*
		or	*case-value* : *case-value*
R815	*case-value*	**is**	*scalar-int-initialization-expr*
		or	*scalar-char-initialization-expr*
		or	*scalar-logical-initialization-expr*

C805 (R808) For a given *case-construct*, each *case-value* shall be of the same type as *case-expr*. For character type, the kind type parameters shall be the same; character length differences are allowed.

C806 (R808) A *case-value-range* using a colon shall not be used if *case-expr* is of type logical.

C807 (R808) For a given *case-construct*, the *case-value-range*s shall not overlap; that is, there shall be no possible value of the *case-expr* that matches more than one *case-value-range*.

R816	*associate-construct*	**is**	*associate-stmt*
			block
			end-associate-stmt
R817	*associate-stmt*	**is**	[*associate-construct-name* :] ASSOCIATE ■
			■ (*association-list*)
R818	*association*	**is**	*associate-name* => *selector*
R819	*selector*	**is**	*expr*
		or	*variable*

C808 (R818) If *selector* is not a *variable* or is a *variable* that has a vector subscript, *associate-name* shall not appear in a variable definition context (16.5.7).

C809 (R818) An *associate-name* shall not be the same as another *associate-name* in the same *associate-stmt*.

R820	*end-associate-stmt*	**is**	END ASSOCIATE [*associate-construct-name*]

C810 (R820) If the *associate-stmt* of an *associate-construct* specifies an *associate-construct-name*, the corresponding *end-associate-stmt* shall specify the same *associate-construct-name*. If the *associate-stmt* of an *associate-construct* does not specify an *associate-construct-name*, the corresponding *end-associate-stmt* shall not specify an *associate-construct-name*.

R821	*select-type-construct*	**is**	*select-type-stmt*
			[*type-guard-stmt*
			block] ...
			end-select-type-stmt
R822	*select-type-stmt*	**is**	[*select-construct-name* :] SELECT TYPE ■
			■ ([*associate-name* =>] *selector*)

C811 (R822) If *selector* is not a named *variable*, *associate-name* => shall appear.

C812 (R822) If *selector* is not a *variable* or is a *variable* that has a vector subscript, *associate-name* shall not appear in a variable definition context (16.5.7).

C813 (R822) The *selector* in a *select-type-stmt* shall be polymorphic.

R823	*type-guard-stmt*	**is**	TYPE IS (*type-spec*) [*select-construct-name*]
		or	CLASS IS (*type-spec*) [*select-construct-name*]

or CLASS DEFAULT [*select-construct-name*]

C814 (R823) The ***type-spec*** shall specify that each length type parameter is assumed.

C815 (R823) The ***type-spec*** shall not specify a sequence derived type or a type with the BIND attribute.

C816 (R823) If ***selector*** is not unlimited polymorphic, the ***type-spec*** shall specify an extension of the declared type of ***selector***.

C817 (R823) For a given ***select-type-construct***, the same type and kind type parameter values shall not be specified in more than one TYPE IS ***type-guard-stmt*** and shall not be specified in more than one CLASS IS ***type-guard-stmt***.

C818 (R823) For a given ***select-type-construct***, there shall be at most one CLASS DEFAULT ***type-guard-stmt***.

R824 *end-select-type-stmt* **is** END SELECT [*select-construct-name*]

C819 (R821) If the ***select-type-stmt*** of a ***select-type-construct*** specifies a ***select-construct-name***, the corresponding ***end-select-type-stmt*** shall specify the same ***select-construct-name***. If the ***select-type-stmt*** of a ***select-type-construct*** does not specify a ***select-construct-name***, the corresponding ***end-select-type-stmt*** shall not specify a ***select-construct-name***. If a ***type-guard-stmt*** specifies a ***select-construct-name***, the corresponding ***select-type-stmt*** shall specify the same ***select-construct-name***.

R825 *do-construct* **is** *block-do-construct*
or *nonblock-do-construct*

R826 *block-do-construct* **is** *do-stmt*
do-block
end-do

R827 *do-stmt* **is** *label-do-stmt*
or *nonlabel-do-stmt*

R828 *label-do-stmt* **is** [*do-construct-name* :] DO *label* [*loop-control*]

R829 *nonlabel-do-stmt* **is** [*do-construct-name* :] DO [*loop-control*]

R830 *loop-control* **is** [,] *do-variable* = *scalar-int-expr* , *scalar-int-expr* ■
■ [, *scalar-int-expr*]
or [,] WHILE (*scalar-logical-expr*)

R831 *do-variable* **is** *scalar-int-variable*

C820 (R831) The ***do-variable*** shall be a named scalar variable of type integer.

R832 *do-block* **is** *block*

R833 *end-do* **is** *end-do-stmt*
or *continue-stmt*

R834 *end-do-stmt* **is** END DO [*do-construct-name*]

C821 (R826) If the ***do-stmt*** of a ***block-do-construct*** specifies a ***do-construct-name***, the corresponding ***end-do*** shall be an ***end-do-stmt*** specifying the same ***do-construct-name***. If the ***do-stmt*** of a ***block-do-construct*** does not specify a ***do-construct-name***, the corresponding ***end-do*** shall not specify a ***do-construct-name***.

C822 (R826) If the ***do-stmt*** is a ***nonlabel-do-stmt***, the corresponding ***end-do*** shall be an ***end-do-stmt***.

C823 (R826) If the ***do-stmt*** is a ***label-do-stmt***, the corresponding ***end-do*** shall be identified with the same ***label***.

R835 *nonblock-do-construct* **is** *action-term-do-construct*
or *outer-shared-do-construct*

R836 *action-term-do-construct* **is** *label-do-stmt*
do-body
do-term-action-stmt

R837 *do-body* **is** [*execution-part-construct*] ...

R838 *do-term-action-stmt* **is** action-stmt

C824 (R838) A *do-term-action-stmt* shall not be a *continue-stmt*, a *goto-stmt*, a *return-stmt*, a *stop-stmt*, an *exit-stmt*, a *cycle-stmt*, an *end-function-stmt*, an *end-subroutine-stmt*, an *end-program-stmt*, or an *arithmetic-if-stmt*.

C825 (R835) The *do-term-action-stmt* shall be identified with a label and the corresponding *label-do-stmt* shall refer to the same label.

R839 *outer-shared-do-construct* **is** *label-do-stmt*
do-body
shared-term-do-construct

R840 *shared-term-do-construct* **is** *outer-shared-do-construct*
or *inner-shared-do-construct*

R841 *inner-shared-do-construct* **is** *label-do-stmt*
do-body
do-term-shared-stmt

R842 *do-term-shared-stmt* **is** action-stmt

C826 (R842) A *do-term-shared-stmt* shall not be a *goto-stmt*, a *return-stmt*, a *stop-stmt*, an *exit-stmt*, a *cycle-stmt*, an *end-function-stmt*, an *end-subroutine-stmt*, an *end-program-stmt*, or an *arithmetic-if-stmt*.

C827 (R840) The *do-term-shared-stmt* shall be identified with a label and all of the *label-do-stmt*s of the *inner-shared-do-construct* and *outer-shared-do-construct* shall refer to the same label.

R843 *cycle-stmt* **is** CYCLE [*do-construct-name*]

C828 (R843) If a *cycle-stmt* refers to a *do-construct-name*, it shall be within the range of that *do-construct*; otherwise, it shall be within the range of at least one *do-construct*.

R844 *exit-stmt* **is** EXIT [*do-construct-name*]

C829 (R844) If an *exit-stmt* refers to a *do-construct-name*, it shall be within the range of that *do-construct*; otherwise, it shall be within the range of at least one *do-construct*.

R845 *goto-stmt* **is** GO TO *label*

C830 (R845) The *label* shall be the statement label of a branch target statement that appears in the same scoping unit as the *goto-stmt*.

R846 *computed-goto-stmt* **is** GO TO (*label-list*) [,] *scalar-int-expr*

C831 (R846 Each *label* in *label-list* shall be the statement label of a branch target statement that appears in the same scoping unit as the *computed-goto-stmt*.

R847 *arithmetic-if-stmt* **is** IF (*scalar-numeric-expr*) *label* , *label* , *label*

C832 (R847) Each *label* shall be the label of a branch target statement that appears in the same scoping unit as the *arithmetic-if-stmt*.

C833 (R847) The *scalar-numeric-expr* shall not be of type complex.

R848 *continue-stmt* **is** CONTINUE

R849 *stop-stmt* **is** STOP [*stop-code*]

R850 *stop-code* **is** *scalar-char-constant*
or *digit* [*digit* [*digit* [*digit* [*digit*]]]]

C834 (R850) *scalar-char-constant* shall be of type default character.

Section 9:

R901 *io-unit* **is** *file-unit-number*
or *
or *internal-file-variable*

R902 *file-unit-number* **is** *scalar-int-expr*

R903	*internal-file-variable*	**is**	*char-variable*

C901 (R903) The *char-variable* shall not be an array section with a vector subscript.

C902 (R903) The *char-variable* shall be of type default character, ASCII character, or ISO 10646 character.

R904	*open-stmt*	**is**	OPEN (*connect-spec-list*)
R905	*connect-spec*	**is**	[UNIT =] *file-unit-number*
		or	ACCESS = *scalar-default-char-expr*
		or	ACTION = *scalar-default-char-expr*
		or	ASYNCHRONOUS = *scalar-default-char-expr*
		or	BLANK = *scalar-default-char-expr*
		or	DECIMAL = *scalar-default-char-expr*
		or	DELIM = *scalar-default-char-expr*
		or	ENCODING = *scalar-default-char-expr*
		or	ERR = *label*
		or	FILE = *file-name-expr*
		or	FORM = *scalar-default-char-expr*
		or	IOMSG = *iomsg-variable*
		or	IOSTAT = *scalar-int-variable*
		or	PAD = *scalar-default-char-expr*
		or	POSITION = *scalar-default-char-expr*
		or	RECL = *scalar-int-expr*
		or	ROUND = *scalar-default-char-expr*
		or	SIGN = *scalar-default-char-expr*
		or	STATUS = *scalar-default-char-expr*
R906	*file-name-expr*	**is**	*scalar-default-char-expr*
R907	*iomsg-variable*	**is**	*scalar-default-char-variable*

C903 (R905) No specifier shall appear more than once in a given *connect-spec-list*.

C904 (R905) A *file-unit-number* shall be specified; if the optional characters UNIT= are omitted, the *file-unit-number* shall be the first item in the *connect-spec-list*.

C905 (R905) The *label* used in the ERR= specifier shall be the statement label of a branch target statement that appears in the same scoping unit as the OPEN statement.

R908	*close-stmt*	**is**	CLOSE (*close-spec-list*)
R909	*close-spec*	**is**	[UNIT =] *file-unit-number*
		or	IOSTAT = *scalar-int-variable*
		or	IOMSG = *iomsg-variable*
		or	ERR = *label*
		or	STATUS = *scalar-default-char-expr*

C906 (R909) No specifier shall appear more than once in a given *close-spec-list*.

C907 (R909) A *file-unit-number* shall be specified; if the optional characters UNIT= are omitted, the *file-unit-number* shall be the first item in the *close-spec-list*.

C908 (R909) The *label* used in the ERR= specifier shall be the statement label of a branch target statement that appears in the same scoping unit as the CLOSE statement.

R910	*read-stmt*	**is**	READ (*io-control-spec-list*) [*input-item-list*]
		or	READ *format* [, *input-item-list*]
R911	*write-stmt*	**is**	WRITE (*io-control-spec-list*) [*output-item-list*]
R912	*print-stmt*	**is**	PRINT *format* [, *output-item-list*]
R913	*io-control-spec*	**is**	[UNIT =] *io-unit*

or [FMT =] *format*
or [NML =] *namelist-group-name*
or ADVANCE = *scalar-default-char-expr*
or ASYNCHRONOUS = *scalar-char-initialization-expr*
or BLANK = *scalar-default-char-expr*
or DECIMAL = *scalar-default-char-expr*
or DELIM = *scalar-default-char-expr*
or END = *label*
or EOR = *label*
or ERR = *label*
or ID = *scalar-int-variable*
or IOMSG = *iomsg-variable*
or IOSTAT = *scalar-int-variable*
or PAD = *scalar-default-char-expr*
or POS = *scalar-int-expr*
or REC = *scalar-int-expr*
or ROUND = *scalar-default-char-expr*
or SIGN = *scalar-default-char-expr*
or SIZE = *scalar-int-variable*

C909 (R913) No specifier shall appear more than once in a given *io-control-spec-list*.

C910 (R913) An *io-unit* shall be specified; if the optional characters UNIT= are omitted, the *io-unit* shall be the first item in the *io-control-spec-list*.

C911 (R913) A DELIM= or SIGN= specifier shall not appear in a *read-stmt*.

C912 (R913) A BLANK=, PAD=, END=, EOR=, or SIZE= specifier shall not appear in a *write-stmt*.

C913 (R913) The *label* in the ERR=, EOR=, or END= specifier shall be the statement label of a branch target statement that appears in the same scoping unit as the data transfer statement.

C914 (R913) A *namelist-group-name* shall be the name of a namelist group.

C915 (R913) A *namelist-group-name* shall not appear if an *input-item-list* or an *output-item-list* appears in the data transfer statement.

C916 (R913) An *io-control-spec-list* shall not contain both a *format* and a *namelist-group-name*.

C917 (R913) If *format* appears without a preceding FMT=, it shall be the second item in the *io-control-spec-list* and the first item shall be *io-unit*.

C918 (R913) If *namelist-group-name* appears without a preceding NML=, it shall be the second item in the *io-control-spec-list* and the first item shall be *io-unit*.

C919 (R913) If *io-unit* is not a *file-unit-number*, the *io-control-spec-list* shall not contain a REC= specifier or a POS= specifier.

C920 (R913) If the REC= specifier appears, an END= specifier shall not appear, a *namelist-group-name* shall not appear, and the *format*, if any, shall not be an asterisk.

C921 (R913) An ADVANCE= specifier may appear only in a formatted sequential or stream input/output statement with explicit format specification (10.1) whose control information list does not contain an *internal-file-variable* as the *io-unit*.

C922 (R913) If an EOR= specifier appears, an ADVANCE= specifier also shall appear.

C923 (R913) If a SIZE= specifier appears, an ADVANCE= specifier also shall appear.

C924 (R913) The *scalar-char-initialization-expr* in an ASYNCHRONOUS= specifier shall be of type default character and shall have the value YES or NO.

C925 (R913) An ASYNCHRONOUS= specifier with a value YES shall not appear unless *io-unit* is a *file-unit-number*.

C926 (R913) If an ID= specifier appears, an ASYNCHRONOUS= specifier with the value YES shall also appear.

C927 (R913) If a POS= specifier appears, the *io-control-spec-list* shall not contain a REC= specifier.

C928 (R913) If a DECIMAL=, BLANK=, PAD=, SIGN=, or ROUND= specifier appears, a *format* or *namelist-group-name* shall also appear.

C929 (R913) If a DELIM= specifier appears, either *format* shall be an asterisk or *namelist-group-name* shall appear.

R914	*format*	**is**	*default-char-expr*
		or	*label*
		or	*

C930 (R914) The *label* shall be the label of a FORMAT statement that appears in the same scoping unit as the statement containing the FMT= specifier.

R915	*input-item*	**is**	*variable*
		or	*io-implied-do*
R916	*output-item*	**is**	*expr*
		or	*io-implied-do*
R917	*io-implied-do*	**is**	(*io-implied-do-object-list* , *io-implied-do-control*)
R918	*io-implied-do-object*	**is**	*input-item*
		or	*output-item*
R919	*io-implied-do-control*	**is**	*do-variable* = *scalar-int-expr* , ■
			■ *scalar-int-expr* [, *scalar-int-expr*]

C931 (R915) A variable that is an *input-item* shall not be a whole assumed-size array.

C932 (R915) A variable that is an *input-item* shall not be a procedure pointer.

C933 (R919) The *do-variable* shall be a named scalar variable of type integer.

C934 (R918) In an *input-item-list*, an *io-implied-do-object* shall be an *input-item*. In an *output-item-list*, an *io-implied-do-object* shall be an *output-item*.

C935 (R916) An expression that is an *output-item* shall not have a value that is a procedure pointer.

R920	*dtv-type-spec*	**is**	TYPE(*derived-type-spec*)
		or	CLASS(*derived-type-spec*)

C936 (R920) If *derived-type-spec* specifies an extensible type, the CLASS keyword shall be used; otherwise, the TYPE keyword shall be used.

C937 (R920) All length type parameters of *derived-type-spec* shall be assumed.

R921	*wait-stmt*	**is**	WAIT (*wait-spec-list*)
R922	*wait-spec*	**is**	[UNIT =] *file-unit-number*
		or	END = *label*
		or	EOR = *label*
		or	ERR = *label*
		or	ID = *scalar-int-variable*
		or	IOMSG = *iomsg-variable*
		or	IOSTAT = *scalar-int-variable*

C938 (R922) No specifier shall appear more than once in a given *wait-spec-list*.

C939 (R922) A *file-unit-number* shall be specified; if the optional characters UNIT= are omitted, the *file-unit-number* shall be the first item in the *wait-spec-list*.

C940 (R922) The *label* in the ERR=, EOR=, or END= specifier shall be the statement label of a branch target statement that appears in the same scoping unit as the WAIT statement.

R923	*backspace-stmt*	**is**	BACKSPACE *file-unit-number*
		or	BACKSPACE (*position-spec-list*)
R924	*endfile-stmt*	**is**	ENDFILE *file-unit-number*

		or	ENDFILE (*position-spec-list*)
R925	*rewind-stmt*	**is**	REWIND *file-unit-number*
		or	REWIND (*position-spec-list*)
R926	*position-spec*	**is**	[UNIT =] *file-unit-number*
		or	IOMSG = *iomsg-variable*
		or	IOSTAT = *scalar-int-variable*
		or	ERR = *label*

C941 (R926) No specifier shall appear more than once in a given *position-spec-list*.

C942 (R926) A *file-unit-number* shall be specified; if the optional characters UNIT= are omitted, the *file-unit-number* shall be the first item in the *position-spec-list*.

C943 (R926) The *label* in the ERR= specifier shall be the statement label of a branch target statement that appears in the same scoping unit as the file positioning statement.

R927	*flush-stmt*	**is**	FLUSH *file-unit-number*
		or	FLUSH (*flush-spec-list*)
R928	*flush-spec*	**is**	[UNIT =] *file-unit-number*
		or	IOSTAT = *scalar-int-variable*
		or	IOMSG = *iomsg-variable*
		or	ERR = *label*

C944 (R928) No specifier shall appear more than once in a given *flush-spec-list*.

C945 (R928) A *file-unit-number* shall be specified; if the optional characters UNIT= are omitted from the unit specifier, the *file-unit-number* shall be the first item in the *flush-spec-list*.

C946 (R928) The *label* in the ERR= specifier shall be the statement label of a branch target statement that appears in the same scoping unit as the flush statement.

R929	*inquire-stmt*	**is**	INQUIRE (*inquire-spec-list*)
		or	INQUIRE (IOLENGTH = *scalar-int-variable*) ■
			■ *output-item-list*
R930	*inquire-spec*	**is**	[UNIT =] *file-unit-number*
		or	FILE = *file-name-expr*
		or	ACCESS = *scalar-default-char-variable*
		or	ACTION = *scalar-default-char-variable*
		or	ASYNCHRONOUS = *scalar-default-char-variable*
		or	BLANK = *scalar-default-char-variable*
		or	DECIMAL = *scalar-default-char-variable*
		or	DELIM = *scalar-default-char-variable*
		or	DIRECT = *scalar-default-char-variable*
		or	ENCODING = *scalar-default-char-variable*
		or	ERR = *label*
		or	EXIST = *scalar-default-logical-variable*
		or	FORM = *scalar-default-char-variable*
		or	FORMATTED = *scalar-default-char-variable*
		or	ID = *scalar-int-variable*
		or	IOMSG = *iomsg-variable*
		or	IOSTAT = *scalar-int-variable*
		or	NAME = *scalar-default-char-variable*
		or	NAMED = *scalar-default-logical-variable*
		or	NEXTREC = *scalar-int-variable*
		or	NUMBER = *scalar-int-variable*

or OPENED = *scalar-default-logical-variable*
or PAD = *scalar-default-char-variable*
or PENDING = *scalar-default-logical-variable*
or POS = *scalar-int-variable*
or POSITION = *scalar-default-char-variable*
or READ = *scalar-default-char-variable*
or READWRITE = *scalar-default-char-variable*
or RECL = *scalar-int-variable*
or ROUND = *scalar-default-char-variable*
or SEQUENTIAL = *scalar-default-char-variable*
or SIGN = *scalar-default-char-variable*
or SIZE = *scalar-int-variable*
or STREAM = *scalar-default-char-variable*
or UNFORMATTED = *scalar-default-char-variable*
or WRITE = *scalar-default-char-variable*

C947 (R930) No specifier shall appear more than once in a given *inquire-spec-list*.

C948 (R930) An *inquire-spec-list* shall contain one FILE= specifier or one UNIT= specifier, but not both.

C949 (R930) In the inquire by unit form of the INQUIRE statement, if the optional characters UNIT= are omitted, the *file-unit-number* shall be the first item in the *inquire-spec-list*.

C950 (R930) If an ID= specifier appears, a PENDING= specifier shall also appear.

Section 10:

R1001 *format-stmt* **is** FORMAT *format-specification*

R1002 *format-specification* **is** ([*format-item-list*])

C1001 (R1001) The *format-stmt* shall be labeled.

C1002 (R1002) The comma used to separate *format-items* in a *format-item-list* may be omitted

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-literal-constant*

C1003 (R1004) *r* shall be positive.

C1004 (R1004) *r* shall not have a kind parameter specified for it.

R1005 *data-edit-desc* **is** I *w* [. *m*]
or B *w* [. *m*]
or O *w* [. *m*]
or Z *w* [. *m*]
or F *w* . *d*
or E *w* . *d* [E *e*]
or EN *w* . *d* [E *e*]
or ES *w* . *d* [E *e*]
or G *w* . *d* [E *e*]
or L *w*
or A [*w*]
or D *w* . *d*
or DT [*char-literal-constant*] [(*v-list*)]

R1006	*w*	**is**	*int-literal-constant*
R1007	*m*	**is**	*int-literal-constant*
R1008	*d*	**is**	*int-literal-constant*
R1009	*e*	**is**	*int-literal-constant*
R1010	*v*	**is**	*signed-int-literal-constant*

C1005 (R1009) *e* shall be positive.

C1006 (R1006) *w* shall be zero or positive for the I, B, O, Z, and F edit descriptors. *w* shall be positive for all other edit descriptors.

C1007 (R1005) *w*, *m*, *d*, *e*, and *v* shall not have kind parameters specified for them.

C1008 (R1005) The *char-literal-constant* in the DT edit descriptor shall not have a kind parameter specified for it.

R1011	*control-edit-desc*	**is**	*position-edit-desc*
		or	[*r*] /
		or	:
		or	*sign-edit-desc*
		or	*k* P
		or	*blank-interp-edit-desc*
		or	*round-edit-desc*
		or	*decimal-edit-desc*
R1012	*k*	**is**	*signed-int-literal-constant*

C1009 (R1012) *k* shall not have a kind parameter specified for it.

R1013	*position-edit-desc*	**is**	T *n*
		or	TL *n*
		or	TR *n*
		or	*n* X
R1014	*n*	**is**	*int-literal-constant*

C1010 (R1014) *n* shall be positive.

C1011 (R1014) *n* shall not have a kind parameter specified for it.

R1015	*sign-edit-desc*	**is**	SS
		or	SP
		or	S
R1016	*blank-interp-edit-desc*	**is**	BN
		or	BZ
R1017	*round-edit-desc*	**is**	RU
		or	RD
		or	RZ
		or	RN
		or	RC
		or	RP
R1018	*decimal-edit-desc*	**is**	DC
		or	DP
R1019	*char-string-edit-desc*	**is**	*char-literal-constant*

C1012 (R1019) The *char-literal-constant* shall not have a kind parameter specified for it.

Section 11:

R1101	*main-program*	**is**	[*program-stmt*]
			[*specification-part*]

[*execution-part*]
[*internal-subprogram-part*]
end-program-stmt

R1102 *program-stmt* **is** PROGRAM *program-name*

R1103 *end-program-stmt* **is** END [PROGRAM [*program-name*]]

C1101 (R1101) In a *main-program*, the *execution-part* shall not contain a RETURN statement or an ENTRY statement.

C1102 (R1101) The *program-name* may be included in the *end-program-stmt* only if the optional *program-stmt* is used and, if included, shall be identical to the *program-name* specified in the *program-stmt*.

C1103 (R1101) An automatic object shall not appear in the *specification-part* (R204) of a main program.

R1104 *module* **is** *module-stmt*
[*specification-part*]
[*module-subprogram-part*]
end-module-stmt

R1105 *module-stmt* **is** MODULE *module-name*

R1106 *end-module-stmt* **is** END [MODULE [*module-name*]]

R1107 *module-subprogram-part* **is** *contains-stmt*
module-subprogram
[*module-subprogram*] ...

R1108 *module-subprogram* **is** *function-subprogram*
or *subroutine-subprogram*

C1104 (R1104) If the *module-name* is specified in the *end-module-stmt*, it shall be identical to the *module-name* specified in the *module-stmt*.

C1105 (R1104) A module *specification-part* shall not contain a *stmt-function-stmt*, an *entry-stmt*, or a *format-stmt*.

C1106 (R1104) An automatic object shall not appear in the *specification-part* of a module.

C1107 (R1104) If an object of a type for which *component-initialization* is specified (R444) appears in the *specification-part* of a module and does not have the ALLOCATABLE or POINTER attribute, the object shall have the SAVE attribute.

R1109 *use-stmt* **is** USE [[, *module-nature*] ::] *module-name* [, *rename-list*]
or USE [[, *module-nature*] ::] *module-name* , ■
■ ONLY : [*only-list*]

R1110 *module-nature* **is** INTRINSIC
or NON_INTRINSIC

R1111 *rename* **is** *local-name* => *use-name*
or OPERATOR (*local-defined-operator*) => ■
■ OPERATOR (*use-defined-operator*)

R1112 *only* **is** *generic-spec*
or *only-use-name*
or *rename*

R1113 *only-use-name* **is** *use-name*

C1108 (R1109) If *module-nature* is INTRINSIC, *module-name* shall be the name of an intrinsic module.

C1109 (R1109) If *module-nature* is NON_INTRINSIC, *module-name* shall be the name of a nonintrinsic module.

C1110 (R1109) A scoping unit shall not access an intrinsic module and a nonintrinsic module of the same name.

C1111 (R1111) OPERATOR(*use-defined-operator*) shall not identify a *generic-binding*.

C1112 (R1112) The *generic-spec* shall not identify a *generic-binding*.

C1113 (R1112) Each *generic-spec* shall be a public entity in the module.

C1114 (R1113) Each *use-name* shall be the name of a public entity in the module.

R1114	*local-defined-operator*	**is**	*defined-unary-op*
		or	*defined-binary-op*
R1115	*use-defined-operator*	**is**	*defined-unary-op*
		or	*defined-binary-op*

C1115 (R1115) Each *use-defined-operator* shall be a public entity in the module.

R1116	*block-data*	**is**	*block-data-stmt*
			[*specification-part*]
			end-block-data-stmt
R1117	*block-data-stmt*	**is**	BLOCK DATA [*block-data-name*]
R1118	*end-block-data-stmt*	**is**	END [BLOCK DATA [*block-data-name*]]

C1116 (R1116) The *block-data-name* shall be included in the *end-block-data-stmt* only if it was provided in the *block-data-stmt* and, if included, shall be identical to the *block-data-name* in the *block-data-stmt*.

C1117 (R1116) A *block-data specification-part* shall contain only derived-type definitions and ASYNCHRONOUS, BIND, COMMON, DATA, DIMENSION, EQUIVALENCE, IMPLICIT, INTRINSIC, PARAMETER, POINTER, SAVE, TARGET, USE, VOLATILE, and type declaration statements.

C1118 (R1116) A type declaration statement in a *block-data specification-part* shall not contain ALLOCATABLE, EXTERNAL, or BIND attribute specifiers.

Section 12:

R1201	*interface-block*	**is**	*interface-stmt*
			[*interface-specification*] ...
			end-interface-stmt
R1202	*interface-specification*	**is**	*interface-body*
		or	*procedure-stmt*
R1203	*interface-stmt*	**is**	INTERFACE [*generic-spec*]
		or	ABSTRACT INTERFACE
R1204	*end-interface-stmt*	**is**	END INTERFACE [*generic-spec*]
R1205	*interface-body*	**is**	*function-stmt*
			[*specification-part*]
			end-function-stmt
		or	*subroutine-stmt*
			[*specification-part*]
			end-subroutine-stmt
R1206	*procedure-stmt*	**is**	[MODULE] PROCEDURE *procedure-name-list*
R1207	*generic-spec*	**is**	*generic-name*
		or	OPERATOR (*defined-operator*)
		or	ASSIGNMENT (=)
		or	*dtio-generic-spec*
R1208	*dtio-generic-spec*	**is**	READ (FORMATTED)
		or	READ (UNFORMATTED)
		or	WRITE (FORMATTED)

		or	WRITE (UNFORMATTED)
R1209	*import-stmt*	**is**	IMPORT [[::] *import-name-list*]

C1201 (R1201) An *interface-block* in a subprogram shall not contain an *interface-body* for a procedure defined by that subprogram.

C1202 (R1201) The *generic-spec* shall be included in the *end-interface-stmt* only if it is provided in the *interface-stmt*. If the *end-interface-stmt* includes *generic-name*, the *interface-stmt* shall specify the same *generic-name*. If the *end-interface-stmt* includes ASSIGNMENT(=), the *interface-stmt* shall specify ASSIGNMENT(=). If the *end-interface-stmt* includes *dtio-generic-spec*, the *interface-stmt* shall specify the same *dtio-generic-spec*. If the *end-interface-stmt* includes OPERATOR(*defined-operator*), the *interface-stmt* shall specify the same *defined-operator*. If one *defined-operator* is .LT., .LE., .GT., .GE., .EQ., or .NE., the other is permitted to be the corresponding operator <, <=, >, >=, ==, or /=.

C1203 (R1203) If the *interface-stmt* is ABSTRACT INTERFACE, then the *function-name* in the *function-stmt* or the *subroutine-name* in the *subroutine-stmt* shall not be the same as a keyword that specifies an intrinsic type.

C1204 (R1202) A *procedure-stmt* is allowed only in an interface block that has a *generic-spec*.

C1205 (R1205) An *interface-body* of a pure procedure shall specify the intents of all dummy arguments except pointer, alternate return, and procedure arguments.

C1206 (R1205) An *interface-body* shall not contain an *entry-stmt*, *data-stmt*, *format-stmt*, or *stmt-function-stmt*.

C1207 (R1206) A *procedure-name* shall have an explicit interface and shall refer to an accessible procedure pointer, external procedure, dummy procedure, or module procedure.

C1208 (R1206) If MODULE appears in a *procedure-stmt*, each *procedure-name* in that statement shall be accessible in the current scope as a module procedure.

C1209 (R1206) A *procedure-name* shall not specify a procedure that is specified previously in any *procedure-stmt* in any accessible interface with the same generic identifier.

C1210 (R1209) The IMPORT statement is allowed only in an *interface-body*.

C1211 (R1209) Each *import-name* shall be the name of an entity in the host scoping unit.

R1210	*external-stmt*	**is**	EXTERNAL [::] *external-name-list*
R1211	*procedure-declaration-stmt*	**is**	PROCEDURE ([*proc-interface*]) ■
			■ [[, *proc-attr-spec*] ... ::] *proc-decl-list*
R1212	*proc-interface*	**is**	*interface-name*
		or	*declaration-type-spec*
R1213	*proc-attr-spec*	**is**	*access-spec*
		or	*proc-language-binding-spec*
		or	INTENT (*intent-spec*)
		or	OPTIONAL
		or	POINTER
		or	SAVE
R1214	*proc-decl*	**is**	*procedure-entity-name* [=> *null-init*]
R1215	*interface-name*	**is**	*name*

C1212 (R1215) The *name* shall be the name of an abstract interface or of a procedure that has an explicit interface. If *name* is declared by a *procedure-declaration-stmt* it shall be previously declared. If *name* denotes an intrinsic procedure it shall be one that is listed in 13.6 and not marked with a bullet (•).

C1213 (R1215) The *name* shall not be the same as a keyword that specifies an intrinsic type.

C1214 If a procedure entity has the INTENT attribute or SAVE attribute, it shall also have the POINTER attribute.

C1215 (R1211) If a *proc-interface* describes an elemental procedure, each *procedure-entity-name* shall specify an external procedure.

C1216 (R1214) If => appears in *proc-decl*, the procedure entity shall have the POINTER attribute.

C1217 (R1211) If *proc-language-binding-spec* with a NAME= is specified, then *proc-decl-list* shall contain exactly one *proc-decl*, which shall neither have the POINTER attribute nor be a dummy procedure.

C1218 (R1211) If *proc-language-binding-spec* is specified, the *proc-interface* shall appear, it shall be an *interface-name*, and *interface-name* shall be declared with a *proc-language-binding-spec*.

R1216 *intrinsic-stmt* **is** INTRINSIC [::] *intrinsic-procedure-name-list*

C1219 (R1216) Each *intrinsic-procedure-name* shall be the name of an intrinsic procedure.

R1217 *function-reference* **is** *procedure-designator* ([*actual-arg-spec-list*])

C1220 (R1217) The *procedure-designator* shall designate a function.

C1221 (R1217) The *actual-arg-spec-list* shall not contain an *alt-return-spec*.

R1218 *call-stmt* **is** CALL *procedure-designator* [([*actual-arg-spec-list*])]

C1222 (R1218) The *procedure-designator* shall designate a subroutine.

R1219 *procedure-designator* **is** *procedure-name*
or *proc-component-ref*
or *data-ref* % *binding-name*

C1223 (R1219) A *procedure-name* shall be the name of a procedure or procedure pointer.

C1224 (R1219) A *binding-name* shall be a binding name (4.5.4) of the declared type of *data-ref*.

R1220 *actual-arg-spec* **is** [*keyword* =] *actual-arg*

R1221 *actual-arg* **is** *expr*
or *variable*
or *procedure-name*
or *proc-component-ref*
or *alt-return-spec*

R1222 *alt-return-spec* **is** * *label*

C1225 (R1220) The *keyword* = shall not appear if the interface of the procedure is implicit in the scoping unit.

C1226 (R1220) The *keyword* = shall not be omitted from an *actual-arg-spec* unless it has been omitted from each preceding *actual-arg-spec* in the argument list.

C1227 (R1220) Each *keyword* shall be the name of a dummy argument in the explicit interface of the procedure.

C1228 (R1221) A nonintrinsic elemental procedure shall not be used as an actual argument.

C1229 (R1221) A *procedure-name* shall be the name of an external procedure, a dummy procedure, a module procedure, a procedure pointer, or a specific intrinsic function that is listed in 13.6 and not marked with a bullet(•).

C1230 (R1221) In a reference to a pure procedure, a *procedure-name actual-arg* shall be the name of a pure procedure (12.6).

C1231 (R1222) The *label* used in the *alt-return-spec* shall be the statement label of a branch target statement that appears in the same scoping unit as the *call-stmt*.

C1232 (R1221) If an actual argument is an array section or an assumed-shape956Td[3.1-450Td[3.1]TJ/F397.97Tf00Td[

[*execution-part*]
[*internal-subprogram-part*]
end-function-stmt

R1224 *function-stmt* **is** [*prefix*] FUNCTION *function-name* ■
■ ([*dummy-arg-name-list*]) [*su x*]

C1234 (R1224) If RESULT is specified, *result-name* shall not be the same as *function-name* and shall not be the same as the *entry-name* in any ENTRY statement in the subprogram.

C1235 (R1224) If RESULT is specified, the *function-name* shall not appear in any specification statement in the scoping unit of the function subprogram.

R1225 *proc-language-binding-spec* **is** *language-binding-spec*

C1236 (R1225) A *proc-language-binding-spec* with a NAME= specifier shall not be specified in the *function-stmt* or *subroutine-stmt* of an interface body for an abstract interface or a dummy procedure.

C1237 (R1225) A *proc-language-binding-spec* shall not be specified for an internal procedure.

C1238 (R1225) If *proc-language-binding-spec* is specified for a procedure, each of the procedure's dummy arguments shall be a nonoptional interoperable variable (15.2.4, 15.2.5) or an interoperable procedure (15.2.6). If *proc-language-binding-spec* is specified for a function, the function result shall be an interoperable variable.

R1226 *dummy-arg-name* **is** *name*

C1239 (R1226) A *dummy-arg-name* shall be the name of a dummy argument.

R1227 *prefix* **is** *prefix-spec* [*prefix-spec*] ...

R1228 *prefix-spec* **is** *declaration-type-spec*
or RECURSIVE
or PURE
or ELEMENTAL

C1240 (R1227) A *prefix* shall contain at most one of each *prefix-spec*.

C1241 (R1227) A *prefix* shall not specify both ELEMENTAL and RECURSIVE.

C1242 (R1227) A *prefix* shall not specify ELEMENTAL if *proc-language-binding-spec* appears in the *function-stmt* or *subroutine-stmt*.

R1229 *su x* **is** *proc-language-binding-spec* [RESULT (*result-name*)]
or RESULT (*result-name*) [*proc-language-binding-spec*]

R1230 *end-function-stmt* **is** END [FUNCTION [*function-name*]]

C1243 (R1230) FUNCTION shall appear in the *end-function-stmt* of an internal or module function.

C1244 (R1223) An internal function subprogram shall not contain an ENTRY statement.

C1245 (R1223) An internal function subprogram shall not contain an *internal-subprogram-part*.

C1246 (R1230) If a *function-name* appears in the *end-function-stmt*, it shall be identical to the *function-name* specified in the *function-stmt*.

R1231 *subroutine-subprogram* **is** *subroutine-stmt*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-subroutine-stmt

R1232 *subroutine-stmt* **is** [*prefix*] SUBROUTINE *subroutine-name* ■
■ [([*dummy-arg-list*]) [*proc-language-binding-spec*]]

C1247 (R1232) The *prefix* of a *subroutine-stmt* shall not contain a *declaration-type-spec*.

R1233 *dummy-arg* **is** *dummy-arg-name*
or *

R1234 *end-subroutine-stmt* **is** END [SUBROUTINE [*subroutine-name*]]

C1248 (R1234) SUBROUTINE shall appear in the *end-subroutine-stmt* of an internal or module subroutine.

C1249 (R1231) An internal subroutine subprogram shall not contain an ENTRY statement.

C1250 (R1231) An internal subroutine subprogram shall not contain an *internal-subprogram-part*.

C1251 (R1234) If a *subroutine-name* appears in the *end-subroutine-stmt*, it shall be identical to the *subroutine-name* specified in the *subroutine-stmt*.

R1235 *entry-stmt* **is** ENTRY *entry-name* [([*dummy-arg-list*]) [*su x*]]

C1252 (R1235) If RESULT is specified, the *entry-name* shall not appear in any specification or type-declaration statement in the scoping unit of the function program.

C1253 (R1235) An *entry-stmt* shall appear only in an *external-subprogram* or *module-subprogram*. An *entry-stmt* shall not appear within an *executable-construct*.

C1254 (R1235) RESULT shall appear only if the *entry-stmt* is in a function subprogram.

C1255 (R1235) Within the subprogram containing the *entry-stmt*, the *entry-name* shall not appear as a dummy argument in the FUNCTION or SUBROUTINE statement or in another ENTRY statement nor shall it appear in an EXTERNAL, INTRINSIC, or PROCEDURE statement.

C1256 (R1235) A *dummy-arg* shall not be an alternate return indicator if the ENTRY statement is in a function subprogram.

C1257 (R1235) If RESULT is specified, *result-name* shall not be the same as the *function-name* in the FUNCTION statement and shall not be the same as the *entry-name* in any ENTRY statement pure.

C1270 If a procedure that is neither an intrinsic procedure nor a statement function is used in a context that requires it to be pure, then its interface shall be explicit in the scope of that use. The interface shall specify that the procedure is pure.

C1271 All internal subprograms in a pure subprogram shall be pure.

C1272 In a pure subprogram any designator with a base object that is in common or accessed by host or use association, is a dummy argument of a pure function, is a dummy argument with INTENT (IN) of a pure subroutine, or an object that is storage associated with any such variable, shall not be used in the following contexts:

C1273 Any procedure referenced in a pure subprogram, including one referenced via a defined operation, assignment, or finalization, shall be pure.

C1274 A pure subprogram shall not contain a *print-stmt*, *open-stmt*, *close-stmt*, *backspace-stmt*, *endfile-stmt*, *rewind-stmt*, *flush-stmt*, *wait-stmt*, or *inquire-stmt*.

C1275 A pure subprogram shall not contain a *read-stmt* or *write-stmt* whose *io-unit* is a *file-unit-number* or *.

C1276 A pure subprogram shall not contain a *stop-stmt*.

C1277 All dummy arguments of an elemental procedure shall be scalar dummy data objects and shall not have the POINTER or ALLOCATABLE attribute.

C1278 The result variable of an elemental function shall be scalar and shall not have the POINTER or ALLOCATABLE attribute.

C1279 In the scoping unit of an elemental subprogram, an object designator with a dummy argument as the base object shall not appear in a *specification-expr* except as the argument to one of the intrinsic functions BIT_SIZE, KIND, LEN, or the numeric inquiry functions (13.5.6).

Section 13:

Section 14:

Section 15:

C1501 (R429) A derived type with the BIND attribute shall not be a SEQUENCE type.

C1502 (R429) A derived type with the BIND attribute shall not have type parameters.

C1503 (R429) A derived type with the BIND attribute shall not have the EXTENDS attribute.

C1504 (R429) A derived type with the BIND attribute shall not have a *type-bound-procedure-part*.

C1505 (R429) Each component of a derived type with the BIND attribute shall be a nonpointer, nonallocatable data component with interoperable type and type parameters.

Section 16:

D.2 Syntax rule cross-reference

R472	*ac-do-variable*	R471, C493, C497
R470	*ac-implied-do*	R469, C497
R471	*ac-implied-do-control*	R470
R466	*ac-spec*	R465
R469	*ac-value*	R466, R470, C494, C495, C496
R519	*access-id*	R518, C548
R508	*access-spec*	R431, R441, R446, R452, R453, R503, C539, R518, R1213
R518	*access-stmt*	R212, C548
R214	*action-stmt*	R213, R807, C802
R836	*action-term-do-construct*	R835
R1221	*actual-arg*	R1220, C1230

R1218	*call-stmt*	R214, C1231
R808	*case-construct*	R213, C803, C805, C807
——	*case-construct-name*	R809, R810, R811, C803
R812	*case-expr*	R809, C805, C806, C807
R813	*case-selector*	R810
R810	*case-stmt*	R808, C803
R815	*case-value*	R814, C805
R814	*case-value-range*	R813, C806, C807
R309	*char-constant*	C303, R850, C834
R725	*char-expr*	C706, R731, R812
R731	*char-initialization-expr*	R509, C540, C712, R815, R913, C924
R426	*char-length*	R425, R442, C444, R504, C504, C520
R427	*char-literal-constant*	R306, R1005, C1008, R1019, C1012
R424	*char-selector*	R403
R1019	*char-string-edit-desc*	R1003
R606	*char-variable*	C605, R903, C901, C902
R909	*close-spec*	R908, C906, C907
R908	*close-stmt*	R214, C1274
——	*common-block-name*	R523, R544, R557
R558	*common-block-object*	R557, C587, C588, C589
R557	*common-stmt*	R212
R421	*complex-literal-constant*	R306
R443	*component-array-spec*	R441, R442, C440, C441
R441	*component-attr-spec*	R440, C436, C447
R459	*component-data-source*	R458
R442	*component-decl*	R440, C446
R439	*component-def-stmt*	R438, C436, C438, C439, C445
R444	*component-initialization*	R442, C446, C447, C1107
——	*component-name*	R442
R438	*component-part*	R429
R458	*component-spec*	R457, C483, C484, C485, C486, C488, C489
R846	*computed-goto-stmt*	R214, C831
R711	*concat-op*	R310, R710
R905	*connect-spec*	R904, C903, C904
R305	*constant*	R308, R309, R532, R610, R701
R534	*constant-subobject*	R532, R533, C566
R1237	*contains-stmt*	R210, R448, R1107
R848	*continue-stmt*	R214, R833, C824
R1011	*control-edit-desc*	R1003
R843	*cycle-stmt*	R214, C824, C826, C828
R1008	*d*	R1005, C1007
R440	*data-component-def-stmt*	R439
R1005	*data-edit-desc*	R1003
R528	*data-i-do-object*	R527, C554, C555, C561
R529	*data-i-do-variable*	R527, C556
R527	*data-implied-do*	R526, R528, C557, C561

R551	*letter-spec*	R550
R702	*level-1-expr*	R704
R706	*level-2-expr*	R706, R710
R710	*level-3-expr*	R710, R712
R712	*level-4-expr*	R714
R717	*level-5-expr*	R717, R722
R306	*literal-constant*	R305
R1114	*local-defined-operator*	R1111
——	*local-name*	R1111
R724	*logical-expr*	C705, R733, R748, R803, R804, R807, R812, R830
R733	*logical-initialization-expr*	C714, R815
R428	*logical-literal-constant*	R306, C703, C704
R604	*logical-variable*	C603
R830	*loop-control*	R828, R829
R512	*lower-bound*	R511, R514, R516
R631	*lower-bound-expr*	R630, R737, R738
R1007	*m*	R1005, C1007
R1101	*main-program*	R202, C1101
R748	*mask-expr*	R743, R745, R749, R754, C733, C734
R749	*masked-elsewhere-stmt*	R744, C730
R1104	*module*	R202
——	*module-name*	R1105, R1106, C1104, R1109, C1108, C1109
R1110	*module-nature*	R1109, C1108, C1109
R1105	*module-stmt*	R1104, C1104
R1108	*module-subprogram*	R1107, C1253
R1107	*module-subprogram-part*	R1104
R708	*mult-op*	R310, R705
R704	*mult-operand*	R704, R705
R1014	*n*	R1013, C1010, C1011
R304	*name*	R102, R215, C301, R307, R505, R545, R602, R1215, C1212, C1213, R1226
R307	*named-constant*	R305, R422, R423, R463, R539
R539	*named-constant-def*	R538
——	*namelist-group-name*	R552, C573, C575, R913, C914, C915, C916, C918, C920, C928, C929
R553	*namelist-group-object*	R552, C574, C575
R552	*namelist-stmt*	R212
R835	*nonblock-do-construct*	R825
R829	*nonlabel-do-stmt*	R827, C822
R718	*not-op*	R310, R714
R507	*null-init*	R444, R506, R532, R1214
R633	*nullify-stmt*	R214
R728	*numeric-expr*	C709, R847, C833
R505	*object-name*	R504, C505, C511, C524, R520, R521, R541, R544, R546, R548, R603
R413	*octal-constant*	R411
R1112	*only*	R1109

R1113	*only-use-name*	R1112
R904	*open-stmt*	R214, C1274
R537	*optional-stmt*	R212
R720	*or-op*	R310, R716
R715	*or-operand*	R715, R716
R839	*outer-shared-do-construct*	R835, R840, C827
R916	*output-item*	R911, R912, C915, R918, C934, C935, R929
R538	*parameter-stmt*	R206, R207
R610	*parent-string*	R609, C608
____	*parent-type-name*	R431, C426
____	*part-name*	R613, C609, C610, C611, C612, C613, C614, C615, C619
R613	*part-ref*	C555, C560, C577, R612, C614, C615, C617, C618
R735	*pointer-assignment-stmt*	R214, C537, R757
R541	*pointer-decl*	R540
R634	*pointer-object*	R633, C634
R540	*pointer-stmt*	R212
R1013	*position-edit-desc*	R1011
R926	*position-spec*	R923, R924, R925, C941, C942
R707	*power-op*	R310, R704
R1227	*prefix*	R1224, C1240, C1241, C1242, R1232, C1247
R1228	*prefix-spec*	R1227, C1240
____	*primaries*	C1260
R701	*primary*	R702
R912	*print-stmt*	R214, C1274
R447	*private-components-stmt*	R432, C454
R432	*private-or-sequence*	R429, C430
R1213	*proc-attr-spec*	R1211
R450	*proc-binding-stmt*	R448
R446	*proc-component-attr-spec*	R445, C448, C449, C452
R445	*proc-component-def-stmt*	R439, C448
R741	*proc-component-ref*	R740, R742, R1219, R1221
R1214	*proc-decl*	R445, R1211, C1216, C1217
____	*proc-entity-name*	R541, C568
R1212	*proc-interface*	R445, R1211, C1215, C1218
R1225	*proc-language-binding-spec*	C530, R1213, C1217, C1218, C1236, C1237, C1238, C1242, R1229, R1232
R545	*proc-pointer-name*	R544, C569, R558, C587, C590, R634, R740
R740	*proc-pointer-object*	R735
R742	*proc-target*	R459, C490, C537, R735, C728
____	*procedure-component-name*	R741, C725
R1211	*procedure-declaration-stmt*	R207, C568, C1212
R1219	*procedure-designator*	R1217, C1220, R1218, C1222
____	*procedure-entity-name*	R1214, C1215
____	*procedure-name*	R451, C456, C457, C458, R742, C727, R1206, C1207, C1208, C1209, R1219, C1223, R1221, C1229, C1230
R1206	*procedure-stmt*	R1202, C1204, C1208, C1209
____	*program-name*	R1102, R1103, C1102

R1102	*program-stmt*	R1101, C1102
R202	*program-unit*	R201
R542	*protected-stmt*	R212
R1004	*r*	R1003, C1003, C1004, R1011
R910	*read-stmt*	R214, C911, C1275
R417	*real-literal-constant*	R306, R416
R422	*real-part*	R421
R713	*rel-op*	R310, R712
R1111	*rename*	R1109, R1112
____	*rep-char*	R427
____	*result-name*	C1234, R1229, C1257
R1236	*return-stmt*	R214, C824, C826, C1258
R925	*rewind-stmt*	R214, C1274
R468	*right-square-bracket*	R465
R1017	*round-edit-desc*	R1011
R543	*save-stmt*	R212
R544	*saved-entity*	R543
R103	*scalar-xyz*	C101
R619	*section-subscript*	R613, C613, C618
R809	*select-case-stmt*	R808, C803
____	*select-construct-name*	R822, R823, R824, C819
R821	*select-type-construct*	R213, C817, C818, C819
R822	*select-type-stmt*	R821, C813, C819
R819	*selector*	R818, C808, R822, C811, C812, C813, C816
R434	*sequence-stmt*	R432
R840	*shared-term-do-construct*	R839
R410	*sign*	R405, R408, R416
R1015	*sign-edit-desc*	R1011
R408	*signed-digit-string*	R420
R405	*signed-int-literal-constant*	R422, R423, R532, R1010, R1012
R416	*signed-real-literal-constant*	R422, R423, R532
R418	*significand*	R417
R627	*source-expr*	R624, C631, C632, C633
____	*special-character*	R301
R451	*specific-binding*	R450
R729	*specification-expr*	C501, C504, R512, R513, C1279
R204	*specification-part*	C459, C539, C548, R1101, C1103, R1104, C1105, C1106, C1107, R1116, C1117, C1118, R1205, R1223, R1231, C1266, C1267, C1268, C1269
R212	*specification-stmt*	R207
R625	*stat-variable*	R624, R636
R1238	*stmt-function-stmt*	R207, C1105, C1206
R850	*stop-code*	R849
R849	*stop-stmt*	R214, C824, C826, C1276
R621	*stride*	R620, R755, C736
R614	*structure-component*	R528, C559, C560, C561, R603, R610, R629, R634
R457	*structure-constructor*	R532, C564, R701

R911	*write-stmt*	R214, C912, C1275

Annex E

(Informative)

Index

In this index, entries in *italics* denote BNF terms, entries in **bold face** denote language keywords, and page numbers in **bold face** denote primary or defining text.

Symbols

B

F

O

P

Q

R