WORKING DRAFT

J3/97-007R2

October 21, 1997 12:28 pm

This is an internal working document of J3.

Contents

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 techical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, 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 third edition cancels and replaces the second edition (ISO/IEC 1539-1:1991), 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 string*s

Annexes A to D 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. With the limitations noted in 1.5.1, and the deletions described in Annex B, the syntax and semantics of Fortran 90 are contained entirely within Fortran 95. Therefore, any standard-conforming Fortran 90 program not containing deleted features or affected by such limitations is a standard conforming Fortran 95 program. New features of Fortran 95 can be compatibly incorporated into such Fortran 90 programs, with any exceptions indicated in the text of this part of the standard.

Fortran 95 continues the evolutionary model introduced in Fortran 90 by deleting several of the features marked as obsolescent in Fortran 90 and identifying a few newly-obsolescent features (Annex B).

Fortran 95 is a relatively minor evolution of standard Fortran, with the emphasis in this revision being upon correcting defects in the Fortran 90 standard, including providing interpretation to a number of questions that have arisen concerning Fortran 90 semantics and syntax (e.g., whether blanks are permitted within edit descriptors in free source form). In addition to such corrections and clarifications, Fortran 95 contains several extensions to Fortran 90; there are three major extensions:

(1) The FORALL statement and construct

(2) PURE and ELEMENTAL procedures

(3) Pointer initialization and structure default initialization

FORALL

The Fortran 90 array constructor and SPREAD and RESHAPE intrinsic functions are powerful tools for element-by-element construction of an array value. Their use in combination, which is required for many array values, can be awkward. Fortran 95 therefore provides a simple and efficient alternative: the FORALL statement allows array elements, array sections, character substrings, or pointer targets to be explicitly specified as a function of the element subscripts. The form of the FORALL statement is very much like a functionally equivalent set of nested DO loops for computing and assigning the elements of an array, except that conceptually all elements are computed simultaneously and then assigned simultaneously. An added benefit of FORALL is that it simplifies conversion from sequential DO loops to parallel array operations. A FORALL construct allows several such array assignments to share the same element subscript control. This control includes masking in a manner similar to the masking facilities of WHERE, the main difference between WHERE and FORALL being that FORALL makes use of element subscripts whereas WHERE is whole array oriented.

PURE

As has always been the case in Fortran, Fortran 95 functions may have side effects (e.g., change the value of an argument or a global variable). Side effects cause problems in parallel processing, however, and because parallel processing has become an important high performance technology, Fortran 95 makes it possible to specify a function to be side effect free. Such a function is called "pure" and is declared with the keyword PURE in the function statement. A restricted form of PURE functions may be called elementally; such ELEMENTAL functions are especially important to high performance parallel processing. An added advantage of pure functions is that it is reasonable to allow them in specification expressions; this provides a significant amount of functionality, with very little cost, and therefore this capability has also been included in Fortran 95.

Initialization

In Fortran 90 there was no way to define the initial pointer association status — a pointer has to be explicitly nullified, allocated, or associated with a target during execution before it can be tested by the ASSOCIATED intrinsic function. This limits the usefulness of pointers, especially the use of pointers as derived-type components. Fortran 95 therefore solves this problem by providing (a) a NULL intrinsic function that may be used to nullify a pointer and (b) a means to specify default initial values for derived-type components. In the latter case the specification of initial values is part of the derived-type definition, and objects declared of this type automatically have all their components so initialized.

Organization of this part of ISO/IEC 1539

This part of ISO/IEC 1539 is organized in 14 sections, dealing with 7 conceptual areas. These 7 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
Execution control	Section 8
Input/output	Sections 9, 10
Program units	Sections 11, 12
Scoping and association rules	Section 14

High/low level concepts

Section 2 (Fortran terms and concepts) contains many of the high level concepts of Fortran. This includes the concept of a program and the relationships among its major parts. Also included are the syntax of program units, the rules for statement ordering, and the definitions of many of the fundamental terms used throughout this part of ISO/IEC 1539.

Section 3 (Characters, lexical tokens, and source form) describes the low level elements of Fortran, such as the character set and the allowable forms for source programs. It also contains the rules for constructing literal constants and names for Fortran entities, and lists all of the Fortran operators.

Data concepts

The array operations and data structures provide a rich set of data concepts in Fortran. The main concepts are those of data type, data object, and the use of data objects, which are described in Sections 4, 5, and 6, respectively.

Section 4 (Intrinsic and derived data types) describes the distinction between a data type and a data object, and then focuses on data type. It defines a data type as a set of data values, corresponding forms (constants) for representing these values, and operations on these values. The concept of an intrinsic data type is introduced, and the properties of Fortran's intrinsic types (integer, real, complex, logical, and character) are described. Note that only type concepts are described here, and not the declaration and properties of data objects.

Section 4 also introduces the concept of derived (user-defined) data types, which are compound types whose components ultimately resolve into intrinsic types. The details of defining a derived type are given (note that this has no counterpart with intrinsic types; intrinsic types are predefined and therefore need not - indeed cannot - be redefined by the programmer). As with intrinsic types, this section deals only with type properties, and not with the declaration of data objects of derived type.

Section 5 (Data object declarations and specifications) describes in detail how named data objects are declared and given the desired properties (attributes). An important attribute (the only one required for each data object) is the data type, so the type declaration statement is the main feature of this section. The various attributes are described in detail, as well as the two ways that attributes may be specified (type declaration statements and attribute specification statements). Implicit typing and storage association (COMMON and EQUIVALENCE) are also described in this section, as well as data object value initialization.

Section 6 (Use of data objects) deals mainly with the concept of a variable, and describes the various forms that variables may take. Scalar variables include character strings and substrings, structured (derived-type) objects, structure components, and array elements. Array variables include whole arrays and array sections. Among the array facilities described here are array operations, allocation and deallocation (user controlled dynamic arrays). New in Fortran 95 is automatic deallocation of allocatable arrays in situations that caused them in Fortran 90 to have undefined allocation status; this decreases potential problems due to allocated memory leaks. Note that this applies only to arrays declared with the ALLOCATABLE attribute - not to pointers.

Computations

Section 7 (Expressions and assignment) describes how computations are expressed in Fortran. This includes the forms that expression operands (primaries) may take and the role of operators in these expressions. Operator precedence is rigorously defined in syntax rules and summarized in tabular form. This description includes the relationship of defined operators (user-defined operators) to the intrinsic operators (+, *, **.**AND**.**, **.**OR**.**, etc.). The rules for both expression evaluation and the interpretation (semantics) of intrinsic and defined operators are described in detail.

Section 7 also describes assignment of computational results to data objects, which has four principal forms: the conventional assignment statement, the pointer assignment statement, the WHERE statement and construct, and the FORALL statement and construct. The WHERE and FORALL statements and constructs allow masked array assignment, the main difference between WHERE and FORALL being that FORALL makes use of element subscripts whereas WHERE is whole array oriented.

Section 13 (Intrinsic procedures) describes more than one hundred intrinsic procedures that provide a rich set of computational capabilities. In addition to the Fortran 90 intrinsic procedures, this includes CPU_TIME, NULL, and extensions to CEILING, FLOOR, MAXLOC, and MINLOC.

Execution control

Section 8 (Execution control) describes the control constructs (IF, CASE, and DO), and the control statements (IF, CONTINUE, GO TO, and STOP).

Input/output

Section 9 (Input/output statements) contains definitions for records, files, file connections (OPEN, CLOSE, and preconnected files), data transfer statements (READ, WRITE, and PRINT) that include processing of partial and variable length records, file positioning (REWIND and BACKSPACE), and file inquiry (INQUIRE).

Section 10 (Input/output editing) describes input/output formatting. This includes the FORMAT statement and FMT= specifier, edit descriptors, list-directed formatting, and namelist formatting.

Program units

Section 11 (Program units) describes main programs, external subprograms, modules, and block data program units. Modules, along with the USE statement, are described as a mechanism for encapsulating data and procedure definitions that are to be used by (accessible to) other program

units. Modules are described as vehicles for defining global derived-type definitions, global data object declarations, procedure libraries, and combinations thereof.

Section 12 (Procedures) contains a comprehensive treatment of procedure definition and invocation, including that for user-defined functions and subroutines. The concepts of implicit and explicit procedure interfaces are explained, and situations requiring explicit procedure interfaces are identified. The rules governing actual and dummy arguments, and their association, are described. PURE procedures and ELEMENTAL procedures (which are PURE procedures that may be called elementally) are free of side effects, thereby facilitating parallel processing.

Section 12 also describes the use of the OPERATOR option in interface blocks to allow function invocation in the form of infix and prefix operators as well as the traditional functional form. Similarly, the use of the ASSIGNMENT option in interface blocks is described as allowing an alternate syntax for certain subroutine calls. This section also contains descriptions of recursive procedures, the RETURN statement, the ENTRY statement, internal procedures and the CONTAINS statement, statement functions, generic procedure names, and the means of accessing non-Fortran procedures.

Scoping and association rules

Section 14 (Scope, association, and definition) explains the use of the term "scope" and describes the scope properties of various entities, including names and operators. Also described are the general rules governing procedure argument association, pointer association, and storage association. Finally, Section 14 describes the events that cause variables to become defined (have predictable values) and events that cause variables to become undefined.

Annexes

Annex A. A glossary of common and important terms used in this part of ISO/IEC 1539.

Annex B. A list of all obsolescent features and descriptions of all deleted features. Obsolescent features are still part of Fortran 95 and are described in the normative portions of this part of ISO/IEC 1539. Deleted features are not part of standard Fortran 95, but they are described completely in this annex for the benefit of those implementations that provide complete backward compatibility with Fortran 90.

Annex C. Long notes of explanation, examples, rationales and other informative material. Wherever feasible such material is integrated into the normative sections of this part of ISO/IEC 1539, but clearly identified as supporting informative material. In those cases in which such informative material is so extensive that it would unduly disrupt the flow of normative discourse, the material is placed in this annex.

Annex D. A comprehensive index to this part of ISO/IEC 1539, including the use of principal terms in the syntax rules.

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. A processor conforming to part 1 need not conform to ISO/IEC 1539-2; however, conformance to ISO/IEC 1539-2 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 specific 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 their associated 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 their associated 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 kind type parameter values (4.3) not supported by the processor;

(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 14;

(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 an interface body or the EXTERNAL attribute in the same scoping unit (14). A standard-conforming program shall not use nonstandard intrinsic procedures 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**, and 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.

1.5.1 Fortran 90 compatibility

Except as noted in this section, this standard is an upward compatible extension to the preceding Fortran International Standard, ISO/IEC 1539:1991, informally referred to as Fortran 90. Any standard-conforming Fortran 90 program that does not use one of the deleted features below remains standard-conforming under this standard. The following features present in Fortran 90 are not present in this standard (B.1):

(1) Real and double precision DO variables,

(2) Branching to an ENDIF statement from outside its IF construct,

(3) PAUSE statement,

(4) ASSIGN and assigned GOTO statements and assigned format specifiers, and

(5) H edit descriptor.

NOTE 1.2

Since a standard-conforming program is permitted to contain only forms and relationships described in this standard, any standard-conforming Fortran 90 program that contains any of these deleted features is not standard-conforming under this standard.

The following Fortran 90 features have different interpretations in this International Standard:

(1) 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.

(2) This standard has more intrinsic procedures than did Fortran 90. Therefore, a standard-conforming Fortran 90 program may have a different interpretation under this standard if it invokes an external procedure having the same name as one of the new standard intrinsic procedures, unless that procedure is specified in an EXTERNAL statement or an interface body.

1.5.2 FORTRAN 77 compatibility

Except as noted in this section, the Fortran 95 Standard is an upward compatible extension to the earlier Fortran International Standard, ISO 1539:1980, informally referred to as FORTRAN 77. Any standard-conforming FORTRAN 77 program that does not use one of the deleted features listed in 1.5.1 remains standard conforming under the Fortran 95 Standard; however, see item (4) below regarding intrinsic procedures. The Fortran 95 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 the Fortran 95 Standard, yet remain a standard-conforming program. The following FORTRAN 77 features have different interpretations in the Fortran 95 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. The Fortran 95 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. The Fortran 95 Standard specifies (5.2.10) that this named variable has the SAVE attribute.

(3) FORTRAN 77 required 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. The Fortran 95 Standard specifies (9.4.4.4.2) 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 statement.

(4) The Fortran 95 Standard has more intrinsic functions than did FORTRAN 77 and adds a few intrinsic subroutines. Therefore, a standard-conforming FORTRAN 77 program may have a different interpretation under the Fortran 95 Standard if it invokes an external procedure having the same name as one of the new standard intrinsic procedures, unless that procedure is specified in an EXTERNAL statement as recommended for nonintrinsic functions in appendix B of the FORTRAN 77 standard.

(5) 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, the Fortran 95 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 a different output form than Fortran 95 processors for certain combinations of values and G edit descriptors.

(6) 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.

1.6 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.

1.6.1 Informative notes

Informative notes of explanation, rationale, examples, and other material are interspersed with the normative body of this standard. The informative material is identified by shading and is non-normative.

1.6.2 Syntax rules

Syntax rules are used to help 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 specific syntactic entities shall be substituted in actual statements.

Common abbreviations used in syntactic terms are:

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

(3) The syntactic metasymbols used are:

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

(4) Each syntax rule is given a unique identifying number of the form 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 automatically generate a Fortran parser; where a syntax rule is incomplete, it is restricted by the corresponding constraints and text.

NOTE 1.3

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

When specific entities are substituted for digit, actual digit strings might be:

```
4
67
1999
10243852
```

1.6.3 Assumed syntax rules

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

xyz-list **is** *xyz* [, *xyz*] ...

xyz-name **is** *name*

scalar-xyz **is** *xyz*

Constraint: *scalar-xyz* shall be scalar.

1.6.4 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* (R723).

(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 R513), and in a few other cases.

(4) When reference is made to a type parameter, including the surrounding parentheses, the suffix "*-selector*" is used. See, for example, "*kind-selector*" (R506) and "*length-selector*" (R508).

(5) The term "*subscript*" (for example, R617, R618, and R619) is used consistently in array definitions.

1.6.5 Text conventions

In the descriptive text, an English word equivalent of a BNF syntactic term is usually used. Specific statement keywords and attributes are identified in the text by the upper-case keyword, e.g., "END statement". Boldface words are used in the text where they are first defined with a specialized meaning. Obsolescent features (1.7) are shown in a distinguishing type size.

NOTE 1.4

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

1.7 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, but are still frequently used.

1.7.1 Nature of deleted features

(1) Better methods existed in FORTRAN 77.

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

1.7.2 Nature of obsolescent features

(1) Better methods existed in Fortran 90.

(2) It is recommended that programmers should 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 (1.6.5).

(4) If the use of these features has become insignificant in Fortran programs, future Fortran standards committees should consider deleting them from the next revision.

(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.

1.8 Modules

This standard provides facilities that encourage the design and use of modular and reusable software. Data and procedure definitions may be organized into program units, called modules, and made available to any other program unit. In addition to global data and procedure library facilities, modules provide a mechanism for defining data abstractions and certain language extensions. Modules are described in 11.3.

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

ISO/IEC 646:1991, *Information technology—ISO 7-bit coded character set for information interchange*.

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 1.6.

NOTE 2.1

Some of the syntax rules in this section are subject to constraints that are given only at the appropriate places in later sections.

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

A *program* shall contain exactly one *main-program program-unit*.

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*

R1216 *function-subprogram* **is** *function-stmt*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-function-stmt

R1221 *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

R1112 *block-data* **is** *block-data-stmt*
[*specification-part*]
end-block-data-stmt

R204 *specification-part* **is** [*use-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 *interface-block*
or *type-declaration-stmt*
or *specification-stmt*
or *parameter-stmt*
or *format-stmt*
or *entry-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 *module-subprogram-part* **is** *contains-stmt*
module-subprogram
[*module-subprogram*] ...

R213 *module-subprogram* **is** *function-subprogram*
or *subroutine-subprogram*

R214 *specification-stmt* **is** *access-stmt*
or *allocatable-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 *save-stmt*
or *target-stmt*

R215 *executable-construct* **is** *action-stmt*
or *case-construct*
or *do-construct*
or *forall-construct*
or *if-construct*

		or	*where-construct*
R216	*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	*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	*where-stmt*
		or	*write-stmt*
		or	*arithmetic-if-stmt*
		or	*computed-goto-stmt*

Constraint: 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 Sections 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 derived-type definition (4.4.1),

(2) A procedure interface body, excluding any derived-type definitions and procedure interface bodies in it (12.3.2.1), or

(3) A program unit or subprogram, excluding derived-type definitions, procedure interface bodies, and subprograms in it.

A scoping unit that immediately surrounds another scoping unit is called the **host scoping unit**.

2.2.1 Program

A **program** consists of exactly one main program unit and any number (including zero) of other kinds of program units. The set of program units may include any combination of the different kinds of program units in any order as long as there is only one main program unit.

NOTE 2.2

There is a restriction that there shall be no more than one unnamed block data program unit (11.4).

Since the public portions of a module are required to be available by the time a module reference (11.3.1) is processed, a processor may require a specific order of processing of the program units.

2.2.2 Main program

The main program is described in 11.1.

2.2.3 Procedure

A **procedure** encapsulates an arbitrary sequence of computations 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 or by a defined assignment statement (12.4, 12.4.3, 7.5.1.3). 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 (R213). A module procedure may be invoked by another module subprogram in the module or by any scoping unit that accesses the module procedure by use association (11.3.2). The module containing the subprogram is called the **host** 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 called the **host** of the internal procedure. An internal procedure is local to its host in the sense that the internal procedure is accessible within the scoping units of the host and all its other internal procedures but is not accessible elsewhere.

2.2.3.4 Procedure interface block

The purpose of a procedure **interface block** is to describe the interfaces (12.3) to a set of procedures and to optionally permit them to be invoked through either a single generic name, a defined operator, or a defined assignment. It determines the forms of reference through which the procedures may be invoked (12.4).

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 procedure interface blocks. The purpose of a module is to make the definitions it contains accessible to all other program units that request access. A scoping unit in another program unit may request access to 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 certain executable statements may appear only in certain executable constructs.

2.3.1 Executable/nonexecutable statements

Program execution is a sequence, in time, of computational actions. An **executable statement** is an instruction to perform or control one or more of these actions. Thus, the executable statements of a program unit determine the computational behavior of the program unit. The executable statements are all of those that make up the syntactic class of *executable-construct*.

Nonexecutable statements do not specify actions; they are used to configure the program environment in which computational actions take place. The nonexecutable statements are all those not classified as executable. All statements in a block data program unit shall be nonexecutable. A module is permitted to contain executable statements only within a subprogram in the module.

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 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,
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>

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 and subprograms. 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 2.2 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
ENTRY statement	No	No	No	Yes	Yes	No	No
FORMAT statement	Yes	No	No	Yes	Yes	Yes	No
Misc. declarations (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, and specification statements.
2) Derived type definitions are also scoping units, but they do not contain any of the above statements, and so have not been listed in the table.
3) 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. Executing an *end-program-stmt* causes termination of execution of the program. Executing an *end-function-stmt* or *end-subroutine-stmt* is equivalent to executing a *return-stmt* in a subprogram.

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

2.3.4 Execution sequence

Execution of a 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, and IF 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.

2.4 Data concepts

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

2.4.1 Data type

A **data type** is 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. This central concept is described in 4.1.

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

2.4.1.1 Intrinsic type

An **intrinsic type** is a type that is defined implicitly, 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 4.3. An intrinsic type may be parameterized, in which case the set of data values depends on the values of the parameters. Such a parameter is called a **type parameter** (4.3). The type parameters are KIND and LEN.

The **kind type parameter** indicates the decimal exponent range for the integer type (4.3.1.1), the decimal precision and exponent range for the real and complex types (4.3.1.2, 4.3.1.3), and the representation methods for the character and logical types (4.3.2.1, 4.3.2.2). 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 implicitly but requires a type definition to declare components of intrinsic or of other derived types. A scalar object of such a derived type is called a **structure** (5.1.1.7). The only intrinsic operation for derived types is assignment with type agreement (7.5.1.5). For each derived type, structure constructors are available to provide values (4.4.4). 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 4.4.

2.4.2 Data value

Each intrinsic type has associated with it a set of values that a datum of that type may take. The values for each intrinsic type are described in 4.3. Because derived types are ultimately specified in terms of components of intrinsic types, the values that objects of a derived type may assume are determined by the type definition and the sets of values of the intrinsic types.

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 data 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.

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 of a named data object may be specified explicitly (5) or implicitly (5.3).

Subobjects are portions of certain named 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), and portions of structures (components). Subobjects are themselves data objects, but subobjects are referenced only by subobject designators. 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 a subobject 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)

2.4.3.1.1 Variable

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

2.4.3.1.2 Constant

A **constant** has a value and cannot become defined or redefined during execution of a program. A constant with a name is called a **named constant** and has the PARAMETER attribute (5.1.2.1). A constant without a name is called a **literal constant** (4.3).

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, and is formed from operands, operators, and parentheses. The type, 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 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, and 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 and the extents may be constant also. However, the extents may vary during execution for a dummy argument array, an automatic array, a pointer array, and an allocatable array.

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-like" 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 (4.5).

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

2.4.6 Pointer

A **pointer** is a variable that has the POINTER attribute. A pointer is **associated** with a target by allocation (6.3.1) or pointer assignment (7.5.2). A pointer shall neither be referenced nor defined until it is associated. A pointer is **disassociated** following execution of a DEALLOCATE or NULLIFY statement, following pointer association with a disassociated pointer, or initially through pointer initialization. A disassociated pointer is not currently associated with a target (14.6.2). If the 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 certain storage constraints (14.6.3).

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 **subobject designator** is a name followed by one or more of the following: component selectors, array section selectors, array element selectors, and substring selectors.

2.5.2 Keyword

The term **keyword** is used in two ways in this standard. A word that is part of the syntax of a statement is a **statement keyword**. These keywords are not reserved words; that is, names with the same spellings are allowed. Examples of statement keywords are: IF, READ, UNIT, KIND, and INTEGER.

An **argument keyword** is a dummy argument name (12.4). Section 13 specifies argument keywords for all of the intrinsic procedures. Argument keywords for external procedures may be specified in a procedure interface block (12.3.2.1).

NOTE 2.5

Argument keywords can make procedure references more readable and allow actual arguments to be in any order. This latter property facilitates use of optional arguments.

2.5.3 Declaration

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

2.5.4 Definition

The term **definition** is used in two ways. First, when a data object is given a valid value during program execution, it is said to become **defined**. This is often accomplished by execution of an assignment statement or input statement. Under certain circumstances, a variable does not have a predictable value and is said to be **undefined**. Section 14 describes the ways in which variables may become defined and undefined. The second use of the term **definition** refers to the declaration of derived types and procedures.

2.5.5 Reference

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

A **procedure reference** is the appearance of the procedure name or its operator symbol or the assignment symbol in a context requiring execution of the procedure at that point.

The appearance of a data object name, data subobject designator, or procedure name in an actual argument list does not constitute a reference to that data object, data subobject, 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 (11.3.1).

2.5.6 Association

Association may be name association (14.6.1), pointer association (14.6.2), or storage association (14.6.3). Name association may be argument association, host association, or use 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.

2.5.7 Intrinsic

The qualifier **intrinsic** signifies that the term to which it is applied is defined in this standard. Intrinsic applies to data types, procedures, assignment statements, and operators. All intrinsic data types, procedures, and operators may be used in any scoping unit without further definition or specification.

2.5.8 Operator

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

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** ("newline", for example)

(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.

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*.

If a processor also permits lower-case letters, the lower-case letters are equivalent to the corresponding upper-case letters in program units except in a character context (3.3).

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 twenty-one **special characters** are shown in Table 3.1.

Table 3.1 Special characters

Character	Name of character	Character	Name of character
	Blank	:	Colon
=	Equals	!	Exclamation point
+	Plus	"	Quotation mark or quote
-	Minus	%	Percent
*	Asterisk	&	Ampersand
/	Slash	;	Semicolon
(	Left parenthesis	<	Less than
)	Right parenthesis	>	Greater than
,	Comma	?	Question mark
.	Decimal point or period	$	Currency symbol
'	Apostrophe		

The twenty-one special characters define the syntactic class *special-character*. The special characters are used for operator symbols, bracketing, and various forms of separating and delimiting other lexical tokens. The special characters $ and ? have no specified use.

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 (4.3.2.1), input/output records (9.1.1), and character string edit descriptors (10.2.1).

The default character type shall support a character set that includes the Fortran character set. Other character sets may be supported by the processor in terms of nondefault character types. The characters available in the nondefault character types are not specified, except that one character in each nondefault character type shall be designated as a blank character to be used as a padding character.

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*] ...

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

NOTE 3.1

Examples of names:

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

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*

Constraint: *int-constant* shall be of type integer.

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

Constraint: *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*

R708 *power-op* **is** **

R709 *mult-op* **is** *
or /

R710 *add-op* **is** +
or −

R712 *concat-op* **is** //

R714 *rel-op* **is** **.**EQ**.**
or **.**NE**.**
or **.**LT**.**
or **.**LE**.**
or **.**GT**.**
or **.**GE**.**
or ==
or /=
or <

or <=
or >
or >=

R719 *not-op* **is** **.**NOT**.**

R720 *and-op* **is** **.**AND**.**

R721 *or-op* **is** **.**OR**.**

R722 *equiv-op* **is** **.**EQV**.**
or **.**NEQV**.**

R311 *defined-operator* **is** *defined-unary-op*
or *defined-binary-op*
or *extended-intrinsic-op*

R704 *defined-unary-op* **is** **.** *letter* [*letter*] ... **.**

R724 *defined-binary-op* **is** **.** *letter* [*letter*] ... **.**

R312 *extended-intrinsic-op* **is** *intrinsic-operator*

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

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*]]]]

Constraint: 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 3.2

For example:

```
99999
10
 010
```

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.

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 (4.3.2.1) 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**, each source line may contain from zero to 132 characters and there are no restrictions on where a statement (or portion of a statement) may appear within a line. However, if a line contains any character that is not of default kind (4.3.2.1), the 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 3.3

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
END BLOCK DATA
END DO
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

NOTE 3.4

Allowing optional blanks at specific places in some keywords (for example, ENDIF or END IF) is intended to permit a reasonable choice to users accustomed to insignificant blanks.

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;
(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.

3.3.1.1 Free form commentary

The character "!" initiates a **comment** except when it appears within a character context. The comment extends to the end of the source line. If the first nonblank character on a line is an "!", the line is called 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. Comments have no effect on the interpretation of the program unit.

NOTE 3.5

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

3.3.1.2 Free form statement separation

The character ";" terminates a statement, except when the ";" appears in a character context or in a comment. This optional termination allows another statement to begin following the ";" on the same line. A ";" shall not appear as the first nonblank character on a line. If a ";" separator is followed by zero or more blanks and one or more ";" separators, the sequence from the first ";" to the last, inclusive, is interpreted as a single ";" separator.

3.3.1.3 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 shall not 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.

3.3.1.3.1 Noncharacter context continuation

If an "&" not in a comment is the last nonblank character on a line or the last nonblank character before an "!", the statement is continued on the next line that is not a comment line. If the first nonblank character on the next noncomment line is an "&", the statement continues at the next character position following the "&"; otherwise, it continues with the first character position of the next noncomment 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.

3.3.1.3.2 Character context continuation

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

3.3.1.4 Free form statements

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

NOTE 3.6

No Fortran statement begins with a digit.

A free form statement shall not have more than 39 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 when 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 comments. Comments may appear anywhere within a program unit and may precede the first statement of the program unit. Comments have no effect on the interpretation of the program unit.

NOTE 3.7

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

3.3.2.2 Fixed form statement separation

The character ";" terminates a statement, except when the ";" appears in a character context, in a comment, or in character position 6. This optional termination allows another statement to begin following the ";" on the same line. A ";" shall not appear as the first nonblank character on a line, except in character position 6. If a ";" separator is followed by zero or more blanks and one or more ";" separators, the sequence from the first ";" to the last, inclusive, is interpreted as a single ";" separator.

3.3.2.3 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 3.8

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 (3.3.2.3).

Comment lines shall not be continued. Comment lines may occur within a continued statement.

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 fixed form statement shall not have more than 19 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.

Section 4: Intrinsic and derived data 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 data type. Each data type has a name. The names of the intrinsic types are predefined by the language; the names of any derived types shall be defined in type definitions (4.4.1). A data type is characterized by a set of values, a means to denote the values, and a set of operations that can manipulate and interpret the values.

NOTE 4.1

For example, the logical data 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 data type is the integer data type. This data 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.

The means by which a value is denoted indicates both the type of the value and a particular member of the set of values characterizing that type. Intrinsic data types are parameterized. In this case, the set of values is constrained by the value of the parameter or parameters. For example, the character data type has a length parameter that constrains the set of character values to those whose length is equal to the value of the parameter.

An intrinsic type is one that is predefined by the language. The intrinsic types are integer, real, complex, character, and logical. The phrase "defined intrinsically" will be used later in this section to mean "predefined" in this sense.

In addition to the intrinsic types, application specific types may be derived. Objects of derived type have **components**. Each component is of an intrinsic type or of a derived type. A type definition (4.4.1) is required to supply the name of the type and the names and types of its components.

NOTE 4.2

For example, if the complex type were not intrinsic but had to be derived, a type definition would be required to supply the name "complex" and declare two components, each of type real. In addition, arithmetic operators would have to be defined.

Means are provided to denote values of a derived type (4.4.4) and to define operations that can be used to manipulate objects of a derived type (4.4.5). A derived type shall be defined in the program, whereas an intrinsic type is predefined.

A derived type may be used only where its definition is accessible (4.4.1). An intrinsic type is always accessible.

4.1 The concept of data type

A data 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.

4.1.1 Set of values

For each data type, there is a set of valid values. The set of valid values may be completely determined, as is the case for logical, or may be determined by a processor-dependent method, as

is the case for integer and real. For complex or derived types, the set of valid values consists of the set of all the combinations of the values of the individual components. For parameterized types, the set of valid values depends on the values of the parameters.

4.1.2 Constants

For each of the intrinsic data types, the syntax for literal constants of that type is specified in this standard. These literal constants are described in 4.3 for each intrinsic type. Within a program, all literal constants that have the same form have the same value.

A constant value may be given a name (5.1.2.1, 5.2.9).

A constant value of derived type may be constructed (4.4.4) using a structure constructor from an appropriate sequence of constant expressions (7.1.6.1). 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 data types, a set of operations and corresponding operators are 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, the only intrinsic operation is assignment. All other operations shall be defined by the program (4.4.5).

4.2 Relationship of types and values to objects

The name of a data 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. Once a derived type is defined, an object may be declared to be of that type. 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 scalar object of derived type is referred to as a structure. The components of a structure are subobjects.

Variables may be objects or subobjects. The data type of a variable determines 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 when 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 (7.5.1.3) and an ASSIGNMENT interface block (12.3.2.1) define an assignment that is not defined intrinsically or redefine an intrinsic derived-type assignment.

NOTE 4.3

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

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

4.3 Intrinsic data types

The intrinsic data types are:

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

NOTE 4.4

In addition to these intrinsic types, this standard provides derived types to allow the creation of new data types. See C.1.1 for an example.

4.3.1 Numeric types

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

4.3.1.1 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 (13.14.52). The decimal exponent range of a representation method is returned by the intrinsic function RANGE (13.14.87). The intrinsic function SELECTED_INT_KIND (13.14.94) 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 is the keyword INTEGER (R502).

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

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

R401 *signed-digit-string* **is** [*sign*] *digit-string*

R402 *digit-string* **is** *digit* [*digit*] ...

R403 *signed-int-literal-constant* **is** [*sign*] *int-literal-constant*

R404 *int-literal-constant* **is** *digit-string* [_ *kind-param*]

R405 *kind-param* **is** *digit-string*
or *scalar-int-constant-name*

R406 *sign* **is** +
or –

Constraint: The value of *kind-param* shall be nonnegative.

Constraint: 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 4.5

Examples of signed integer literal constants are:

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

where SHORT is a scalar integer named constant.

In a DATA statement (5.2.10), an unsigned binary, octal, or hexadecimal literal constant shall correspond to an integer scalar variable.

R407 *boz-literal-constant* **is** *binary-constant*
or *octal-constant*
or *hex-constant*

Constraint: A *boz-literal-constant* may appear only in a DATA statement.

R408 *binary-constant* **is** B ' *digit* [*digit*] ... '
or B " *digit* [*digit*] ... "

Constraint: *digit* shall have one of the values 0 or 1.

R409 *octal-constant* **is** O ' *digit* [*digit*] ... '
or O " *digit* [*digit*] ... "

Constraint: *digit* shall have one of the values 0 through 7.

R410 *hex-constant* **is** Z ' *hex-digit* [*hex-digit*] ... '
or Z " *hex-digit* [*hex-digit*] ... "

R411 *hex-digit* **is** *digit*
or A
or B
or C
or D
or E
or F

In these constants, the binary, octal, and hexadecimal digits are interpreted according to their respective number systems. If the processor supports lower-case letters in the source form, the *hex-digit*s A through F may be represented by their lower-case equivalents.

4.3.1.2 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 (13.14.52). The decimal precision and decimal exponent range of an approximation method are returned by the intrinsic functions PRECISION (13.14.81) and RANGE (13.14.87). The intrinsic function SELECTED_REAL_KIND (13.14.95) returns a kind value based on specified precision and decimal range requirements.

NOTE 4.6

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 SIGN, and

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

NOTE 4.7

On a processor that can distinguish between 0.0 and -0.0,

```
( X .GE. 0.0 )
```

evaluates to **.**TRUE**.** if X = 0.0 or if X = -0.0,

```
( X .LT. 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.

NOTE 4.8

Historically some systems had a distinct negative zero value that presented some difficulties. Fortran standards were specified such that these difficulties had to be handled by the processor and not the user. ANSI/IEEE 754-1985, IEEE standard for binary floating point arithmetic, introduced a negative zero with specific properties. For example when the exact result of an operation is negative but rounding produces a zero, the IEEE 754 specified value is -0.0. This standard includes adjustments intended to permit IEEE 754 compliant processors to behave in accordance with that standard without violating this standard.

The type specifier for the real type is the keyword REAL and the type specifier for the double precision real type is the keyword DOUBLE PRECISION (R502).

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

R412 *signed-real-literal-constant* **is** [*sign*] *real-literal-constant*

R413 *real-literal-constant* **is** *significand* [*exponent-letter exponent*] [_ *kind-param*]
or *digit-string exponent-letter exponent* [_ *kind-param*]

R414 *significand* **is** *digit-string* **.** [*digit-string*]
or **.** *digit-string*

R415 *exponent-letter* **is** E
or D

R416 *exponent* **is** *signed-digit-string*

Constraint: If both *kind-param* and *exponent-letter* are present, *exponent-letter* shall be E.

Constraint: 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
0.45E-4
10.93E7_QUAD
.123
3E4
```

where QUAD is a scalar integer named constant.

4.3.1.3 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 (13.14.52).

The type specifier for the complex type is the keyword COMPLEX (R502). 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 data entity is of type **default complex**.

R417 *complex-literal-constant* **is** (*real-part* , *imag-part*)

R418 *real-part* **is** *signed-int-literal-constant*
or *signed-real-literal-constant*

R419 *imag-part* **is** *signed-int-literal-constant*
or *signed-real-literal-constant*

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 signed integer literal constants, they are converted to the default real approximation method and the constant is of type default complex. If only one of the parts is a signed integer literal constant, the signed integer literal constant is converted to the approximation method selected for the signed real literal constant and the kind type parameter value of the complex literal constant is that of the signed real literal constant.

NOTE 4.10

Examples of complex literal constants are:

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

4.3.2 Nonnumeric types

The **nonnumeric types** are provided for nonnumeric processing. The intrinsic operations defined for each of these types are given below.

4.3.2.1 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 (13.14.52). Any character of a particular representation method representable in the processor may occur in a character string of that representation method.

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

The type specifier for the character type is the keyword CHARACTER (R502).

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

R420 *char-literal-constant* **is** [*kind-param* _] ' [*rep-char*] ... '
or [*kind-param* _] " [*rep-char*] ... "

Constraint: 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 one of the following:

(1) Any character in the processor-dependent character set in fixed source form. A processor may restrict the occurrence of some or all of the control characters.

(2) Any graphic character in the processor-dependent character set in free source form.

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 may 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

Examples of nondefault character literal constants, where the processor supports the corresponding character sets, are:

```
BOLD_FACE_' This is in bold face  '
ITALICS_'This is in italics '
```

where BOLD_FACE and ITALICS are named constants whose values are the kind type parameters for bold face and italic characters, respectively.

4.3.2.1.1 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 (13.14.19) and ICHAR (13.14.45) 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 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'), if the processor supports lower-case letters.

(5) ICHAR (' ') < ICHAR ('0') < ICHAR ('9') < ICHAR ('a') or
ICHAR (' ') < ICHAR ('a') < ICHAR ('z') < ICHAR ('0'), if the processor supports lower-case letters.

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

ISO/IEC 646:1991 (International Reference Version) assigns numerical codes to a set of characters that includes the letters, digits, underscore, and special characters; the sequence of such codes is called in this standard the **ASCII collating sequence**.

NOTE 4.15

ISO/IEC 646:1991 is the international equivalent of ANSI X3.4-1986, commonly known as ASCII. The intrinsic functions ACHAR (13.14.2) and IACHAR (13.14.40) provide conversions between these characters and the integers of the ASCII collating sequence.

The intrinsic functions LGT, LGE, LLE, and LLT (13.14.56-13.14.59) 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.

4.3.2.2 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 (13.14.52).

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

R421 *logical-literal-constant* **is** **.**TRUE**.** [_ *kind-param*]
or **.**FALSE**.** [_ *kind-param*]

Constraint: 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.

The type specifier for the logical type is the keyword LOGICAL (R502).

4.4 Derived types

Additional data types may be derived from the intrinsic data types. A type definition is required to define the name of the type and the names and types of its components. The **direct components** of a derived type are

(1) The components of that type and

(2) For any nonpointer component that is of derived type, the direct components of that derived type.

Ultimately, a derived type is resolved into **ultimate components** that are either of intrinsic type or are pointers.

NOTE 4.16

See C.1.1 for an example

By default, derived types defined in the specification part of a module are accessible (5.1.2.2, 5.2.3) in any scoping unit that accesses the module. This default may be changed to restrict the accessibility of such types to the host module itself. A particular type definition may be declared to be public or private regardless of the default accessibility declared for the module. In addition, a type may be accessible while its components are private.

By default, no storage sequence is implied by the order of the component definitions. However, if the definition of a derived type contains a SEQUENCE statement, 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.

Default initialization is specified for a component of an object of derived type when initialization appears in the component declaration. The object will be initialized as specified in the derived-type definition (14.7.3, 14.7.5) even if the definition is private or inaccessible. Default initialization applies to dummy arguments with INTENT (OUT). Unlike explicit initialization, default initialization (4.4.1) does not imply that the object has the SAVE attribute. If a component has default initialization, it is not required that default initialization be specified for other components of the derived type.

The type specifier for derived types is the keyword TYPE followed by the name of the type in parentheses (R502).

4.4.1 Derived-type definition

R422 *derived-type-def* **is** *derived-type-stmt*
[*private-sequence-stmt*] ...
component-def-stmt
[*component-def-stmt*] ...
end-type-stmt

R423 *derived-type-stmt* **is** TYPE [[, *access-spec*] ::] *type-name*

R424 *private-sequence-stmt* **is** PRIVATE
or SEQUENCE

Constraint: An *access-spec* (5.1.2.2) or a PRIVATE statement within the definition is permitted only if the type definition is within the specification part of a module.

Constraint: A derived type *type-name* shall not be the same as the name of any intrinsic type defined in this standard nor the same as any other accessible derived type *type-name*.

Constraint: The same *private-sequence-stmt* shall not appear more than once in a given *derived-type-def*.

Constraint: If SEQUENCE is present, all derived types specified in component definitions shall be sequence types.

R425 *component-def-stmt* **is** *type-spec* [[, *component-attr-spec-list*] ::] *component-decl-list*

R426 *component-attr-spec* **is** POINTER
or DIMENSION (*component-array-spec*)

R427 *component-array-spec* **is** *explicit-shape-spec-list*
or *deferred-shape-spec-list*

Constraint: If a component of a derived type is of a type declared to be private, either the derived-type definition shall contain the PRIVATE statement or the derived type shall be private.

Constraint: No *component-attr-spec* shall appear more than once in a given *component-def-stmt*.

Constraint: If the POINTER attribute is not specified for a component, a *type-spec* in the *component-def-stmt* shall specify an intrinsic type or a previously defined derived type.

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

Constraint: If the POINTER attribute is specified, each *component-array-spec* shall be a *deferred-shape-spec-list*.

Constraint: If the POINTER attribute is not specified, each *component-array-spec* shall be an *explicit-shape-spec-list*.

Constraint: Each bound in the *explicit-shape-spec* shall be a constant specification expression (7.1.6.2).

R428 *component-decl* **is** *component-name* [(*component-array-spec*)] ■
■ [* *char-length*] [*component-initialization*]

R429 *component-initialization* **is** = *initialization-expr*
or => NULL ( )

Constraint: The * *char-length* option is permitted only if the type specified is character.

Constraint: The character length specified by the *char-length* in a *component-decl* or the *char-selector* in a *type-spec* (5.1, 5.1.1.5) shall be a constant specification expression (7.1.6.2).

Constraint: If *component-initialization* appears, a double colon separator shall appear before the *component-decl-list*.

Constraint: If => appears in *component-initialization*, the POINTER attribute shall appear in the *component-attr-spec-list*. If = appears in *component-initialization*, the POINTER attribute shall not appear in the *component-attr-spec-list*.

R430 *end-type-stmt* **is** END TYPE [*type-name*]

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

NOTE 4.17

The double colon separator in a *component-def-stmt* is required only if a *component-attr-spec* or *component-initialization* is specified; otherwise, it is optional.

If the **SEQUENCE statement** is present, the type is a sequence type. If all of the ultimate components are of type default integer, default real, double precision real, default complex, or default logical and are not pointers, the type is a **numeric sequence type**. If all of the ultimate components are of type default character and are not pointers, the type is a **character sequence type**.

NOTE 4.18

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 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 since 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, 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 non-SEQUENCE structure in memory as best suited to the particular architecture.

If *initialization-expr* appears for a nonpointer component, that component in any object of the type is initially defined or becomes defined as specified in (14.7.5) with the value determined from *initialization-expr.* The *initialization-expr* is evaluated in the scoping unit of the type definition. The evaluation rules are those that would be in effect for intrinsic assignment (7.5.1.4) if *component-name* were a variable assigned the value of *initialization-expr*. If *component-name* 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 4.28). Explicit initialization in a type declaration statement (5.1) overrides default initialization (see Note 4.27).

A component is an array if its *component-decl* contains a *component-array-spec* or its *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, 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 4.19

Default initialization of an array component may be specified by a constant expression consisting of an array constructor (4.5), or of a single scalar that becomes the value of each array element.

A component is a pointer if its *component-attr-spec-list* contains the POINTER attribute. Pointers have an association status of associated, disassociated, or undefined. If no default initialization is specified, the initial status is undefined. To specify that the default initial status of a pointer component is to be disassociated, the pointer assignment symbol (=>) shall be followed by a reference to the intrinsic function NULL ( ) with no argument. No mechanism is provided to specify a default initial status of associated.

The accessibility of a derived type may be declared explicitly by an *access-spec* in its *derived-type-stmt* or in an *access-stmt* (5.2.3). The accessibility is the default if it is not declared explicitly. If a type definition is private, then the type name, the structure constructor (4.4.4) for the type, any entity that is of the type, and any procedure that has a dummy argument or function result that is of the type are accessible only within the module containing the definition.

If a type definition contains a PRIVATE statement, the component names for the type are accessible only within the module containing the definition, even if the type itself is public (5.1.2.2). The component names and hence the internal structure of the type are inaccessible in any scoping unit accessing the module via a USE statement. Similarly, the structure constructor for such a type shall be employed only within the defining module.

NOTE 4.20

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
```

NOTE 4.21

A type definition may have a component that is an array. For example:

```
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 4.22

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.

A derived-type definition may have a component that is of a derived type. For example:

```
TYPE TRIANGLE
   TYPE (POINT) :: A, B, C
END TYPE TRIANGLE
```

An example of declaring a variable T to be of type TRIANGLE is:

```
TYPE (TRIANGLE) :: T
```

NOTE 4.23

An example of a private type 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.

NOTE 4.24

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 4.25

A derived type may have a component that is a pointer. For example:

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

Any object of type REFERENCE will have the four components VOLUME, YEAR, PAGE, and TITLE, plus a pointer to an array of characters holding ABSTRACT. 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 in a pointer assignment statement (7.5.2).

NOTE 4.26

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
   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.1.3 for an example.

NOTE 4.27

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 4.28

The default initial value of a component of derived type may be overridden by default initialization specified in the definition of the type.

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

NOTE 4.28 *(Continued)*

```
TYPE(SINGLE_SCORE) SETUP
```

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

NOTE 4.29

Arrays of structures may be declared with elements that are partially or totally initialized by default.

For example:

```
TYPE MEMBER
    CHARACTER (LEN = 20) NAME
    INTEGER :: TEAM_NO, HANDICAP = 0
    TYPE (SINGLE_SCORE), POINTER :: HISTORY => NULL ( )
END TYPE MEMBER
TYPE (MEMBER) LEAGUE (36)           ! Array of partially initialized elements
TYPE (MEMBER) :: 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.
```

4.4.2 Determination of derived types

A particular type name shall be defined at most once in a scoping unit. 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 have the same name, have the SEQUENCE property, and have components that do not have PRIVATE accessibility and agree in order, name, and attributes. Otherwise, they are of different derived types. A data entity declared using a type with the SEQUENCE property is not of the same type as an entity of a type declared to be PRIVATE or which has components that are PRIVATE.

NOTE 4.30

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 4.31

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

```
PROGRAM PGM
   TYPE EMPLOYEE
      SEQUENCE
      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 4.32

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.

4.4.3 Derived-type values

The set of values of a specific derived type consists of all possible sequences of component values consistent with the definition of that derived type.

4.4.4 Construction of derived-type values

A derived-type definition implicitly defines a corresponding **structure constructor** that allows a scalar value of derived type to be constructed from a sequence of values, one value for each component of the derived type.

R431 *structure-constructor* **is** *type-name* (*expr-list*)

The sequence of expressions in a structure constructor specifies component values that shall agree in number and order with the components of the derived type. If necessary, each value is converted according to the rules of intrinsic assignment (7.5.1.4) to a value that agrees in type and type parameters with the corresponding component of the derived type. For nonpointer components, the shape of the expression shall conform with the shape of the component. A structure constructor whose component values are all constant expressions is a derived-type constant expression. A structure constructor shall not appear before the referenced type is defined.

NOTE 4.33

This example illustrates a derived-type constant expression using a derived type defined in Note 4.20:

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

A derived-type definition may have a component that is an array. Also, an object may be an array of derived type. Such arrays may be constructed using an array constructor (4.5).

Where a component in the derived type is a pointer, the corresponding constructor expression shall evaluate to an object that would be an allowable target for such a pointer in a pointer assignment statement (7.5.2).

NOTE 4.34

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 4.25

```
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 ABSTRACT of the object BIBLIO with the target object TEXT.

4.4.5 Derived-type operations and assignment

Any operation on derived-type entities or nonintrinsic assignment for derived-type entities shall be defined explicitly by a function or a subroutine and a procedure interface block (12.3.2.1). Arguments and function values may be of any derived or intrinsic type.

4.5 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.

R432 *array-constructor* **is** (/ *ac-value-list* /)

R433 *ac-value* **is** *expr*
or *ac-implied-do*

R434 *ac-implied-do* **is** (*ac-value-list* , *ac-implied-do-control*)

R435 *ac-implied-do-control* **is** *ac-do-variable* = *scalar-int-expr* , *scalar-int-expr* ■
■ [, *scalar-int-expr*]

R436 *ac-do-variable* **is** *scalar-int-variable*

Constraint: *ac-do-variable* shall be a named variable.

Constraint: Each *ac-value* expression in the *array-constructor* shall have the same type and kind type parameter.

If the *ac-value* expressions are of type character, each *ac-value* expression in the *array-constructor* shall have the same character length parameter.

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 an *ac-value* sequence under the control of the *ac-do-variable*, as in the DO construct (8.1.4.4).

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

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

The type and type parameters of an array constructor are those of the *ac-value* expressions.

If every expression in an array constructor is a constant expression, the array constructor is a constant expression.

NOTE 4.35

An example is:

```
REAL :: X (3)
X = (/ 3.2, 4.01, 6.5 /)
```

NOTE 4.36

A one-dimensional array may be reshaped into any allowable array shape using the RESHAPE intrinsic function (13.14.90). An example is:

```
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 4.37

Examples of array constructors containing an implied-DO are:

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

and

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

NOTE 4.38

Using the type definition for PERSON in Note 4.20, an example of the construction of a derived-type array value is:

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

NOTE 4.39

Using the type definition for LINE in Note 4.21, 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.

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. A named data object shall not be explicitly specified to have a particular attribute more than once in a scoping unit.

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** *type-spec* [[, *attr-spec*] ... ::] *entity-decl-list*

R502 *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*]
or TYPE (*type-name*)

R503 *attr-spec*
is PARAMETER
or *access-spec*
or ALLOCATABLE
or DIMENSION (*array-spec*)
or EXTERNAL
or INTENT (*intent-spec*)
or INTRINSIC
or OPTIONAL
or POINTER
or SAVE
or TARGET

R504 *entity-decl*
is *object-name* [(*array-spec*)] [* *char-length*] [*initialization*]
or *function-name* [* *char-length*]

R505 *initialization*
is = *initialization-expr*
or => NULL ( )

R506 *kind-selector* **is** ([KIND =] *scalar-int-initialization-expr*)

Constraint: The same *attr-spec* shall not appear more than once in a given *type-declaration-stmt*.

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

Constraint: The ALLOCATABLE attribute may be used only when declaring an array that is not a dummy argument or a function result.

Constraint: An array declared with a POINTER or an ALLOCATABLE attribute shall be specified with an *array-spec* that is a *deferred-shape-spec-list* (5.1.2.4.3).

Constraint: An *array-spec* for an *object-name* that is a function result that does not have the POINTER attribute shall be an *explicit-shape-spec-list*.

Constraint: If the POINTER attribute is specified, the TARGET, INTENT, EXTERNAL, or INTRINSIC attribute shall not be specified.

Constraint: If the TARGET attribute is specified, the POINTER, EXTERNAL, INTRINSIC, or PARAMETER attribute shall not be specified.

Constraint: The PARAMETER attribute shall not be specified for dummy arguments, pointers, allocatable arrays, functions, or objects in a common block.

Constraint: The INTENT and OPTIONAL attributes may be specified only for dummy arguments.

Constraint: An entity shall not have the PUBLIC attribute if its type has the PRIVATE attribute.

Constraint: 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.

Constraint: An entity shall not have the EXTERNAL attribute if it has the INTRINSIC attribute.

Constraint: An entity in an *entity-decl-list* shall not have the EXTERNAL or INTRINSIC attribute specified unless it is a function.

Constraint: An array shall not have both the ALLOCATABLE attribute and the POINTER attribute.

Constraint: The * *char-length* option is permitted only if the type specified is character.

Constraint: The *function-name* shall be the name of an external function, an intrinsic function, a function dummy procedure, or a statement function.

Constraint: The *initialization* shall appear if the statement contains a PARAMETER attribute (5.1.2.1).

Constraint: If *initialization* appears, a double colon separator shall appear before the *entity-decl-list*.

Constraint: *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 array, an external name, an intrinsic name, or an automatic object.

Constraint: If => appears in *initialization*, the object shall have the POINTER attribute. If = appears in *initialization*, the object shall not have the POINTER attribute.

Constraint: The value of *scalar-int-initialization-expr* in *kind-selector* shall be nonnegative and shall specify a representation method that exists on the processor.

NOTE 5.2

The double colon separator in a *type-declaration-stmt* is required only if an *attr-spec* or *initialization* is specified; otherwise, the separator is optional.

A name that identifies a specific intrinsic function in a scoping unit has a type as specified in 13.13. 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 attribute.

The *specification-expr* (7.1.6.2) of a *char-len-param-value* (5.1.1.5) or an *array-spec* (5.1.2.4) may be a nonconstant expression provided the specification expression is in an interface body (12.3.2.1), the

specification part of a subprogram, or the *type-spec* of a FUNCTION statement (12.5.2.2). If a *specification-expr* involves a reference to a specification function (7.1.6.2), the expression is considered to be a nonconstant expression. If the data object being declared depends on the value of such a nonconstant expression and 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 *length-selector* (5.1.1.5) is a nonconstant expression, the length is declared at the entry of 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* becomes defined with the value determined from *initialization-expr* in accordance with the rules of intrinsic assignment (7.5.1.4). 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 ( ), *object-name* shall be a pointer, and its initial association status is disassociated. Use of =>NULL ( ) in a scoping unit is a reference to the intrinsic function NULL.

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, or by inclusion of the *object-name* in a SAVE statement (5.2.4).

NOTE 5.4

Examples of type declaration statements are:

```
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
TYPE (PERSON) :: CHAIRMAN
TYPE(NODE), POINTER :: HEAD => NULL ( )
```

NOTE 5.5

Type declaration statements in FORTRAN 77 required different attributes of an entity to be specified in different statements (INTEGER, SAVE, DATA, etc.). This standard allows the attributes of an entity to be specified in a single extended form of the type statement. For example,

```
INTEGER, DIMENSION (10, 10), SAVE :: A, B, C
REAL, PARAMETER :: PI = 3.14159265, E = 2.718281828
```

To retain compatibility and consistency with FORTRAN 77, most of the attributes that may be specified in the extended type statement may alternatively be specified in separate statements.

5.1.1 Type specifiers

The **type specifier** 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 would otherwise be indicated by the first letter of an entity name (5.3).

5.1.1.1 INTEGER

The INTEGER type specifier is used to declare entities of intrinsic type integer (4.3.1.1). The kind selector, if present, specifies the integer representation method. If the kind selector is absent, the kind type parameter is KIND (0) and the entities declared are of type default integer.

5.1.1.2 REAL

The REAL type specifier is used to declare entities of intrinsic type real (4.3.1.2). The kind selector, if present, specifies the real approximation method. If the kind selector is absent, the kind type parameter is KIND (0.0) and the entities declared are of type default real.

5.1.1.3 DOUBLE PRECISION

The DOUBLE PRECISION type specifier is used to declare entities of intrinsic type double precision real (4.3.1.2). The kind parameter value is KIND (0.0D0). An entity declared with a type specifier REAL (KIND (0.0D0)) is of the same kind as one declared with the type specifier DOUBLE PRECISION.

5.1.1.4 COMPLEX

The COMPLEX type specifier is used to declare entities of intrinsic type complex (4.3.1.3). The kind selector, if present, specifies the real approximation method of the two real values making up the real and imaginary parts of the complex value. If the kind selector is absent, the kind type parameter is KIND (0.0) and the entities declared are of type default complex.

5.1.1.5 CHARACTER

The CHARACTER type specifier is used to declare entities of intrinsic type character (4.3.2.1).

R507 *char-selector* **is** *length-selector*
or (LEN = *char-len-param-value* , ■
■ KIND = *scalar-int-initialization-expr*)
or (*char-len-param-value* , ■
■ [KIND =] *scalar-int-initialization-expr*)
or (KIND = *scalar-int-initialization-expr* ■
■ [, LEN = *char-len-param-value*])

R508 *length-selector* **is** ([LEN =] *char-len-param-value*)
or * *char-length* [,]

R509 *char-length* **is** (*char-len-param-value*)
or *scalar-int-literal-constant*

R510 *char-len-param-value* **is** *specification-expr*
or *

Constraint: The value of *scalar-int-initialization-expr* shall be nonnegative and shall specify a representation method that exists on the processor.

Constraint: The *scalar-int-literal-constant* shall not include a *kind-param*.

Constraint: A function name shall not be declared with an asterisk *char-len-param-value* unless it is the name of the result of an external function or the name of a dummy function.

Constraint: A function name declared with an asterisk *char-len-param-value* shall not be array-valued, pointer-valued, recursive, or pure.

Constraint: The optional comma in a *length-selector* is permitted only in a *type-spec* in a *type-declaration-stmt*.

Constraint: The optional comma in a *length-selector* is permitted only if no double colon separator appears in the *type-declaration-stmt*.

The *char-selector* in a CHARACTER *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 *char-len-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.

An **assumed character length parameter** is a type parameter of a dummy argument that is specified with an asterisk *char-len-param-value*.

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 * may be used only in the following ways:

(1) It may be used to declare a dummy argument of a procedure, in which case the dummy argument assumes the length of the associated actual argument when the procedure is invoked.

(2) It may be used to declare a named constant, in which case the length is that of the constant value.

(3) In an external function, the name of the function result may be specified with a character length parameter value of *; in this case, 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.

NOTE 5.6

An interface body may be specified for a dummy or external function whose result has a character length parameter of * only if the function is not invoked. This is because this characteristic has to be specified to be the same in the interface body as in the procedure definition, but in order to invoke such a procedure, the calling routine is required to specify a length other than *.

The length specified for a character-valued statement function or statement function dummy argument of type character shall be a constant specification expression.

The kind selector, if present, specifies the character representation method. If the kind selector is absent, the kind type parameter is KIND ('A') and the entities declared are of type default character.

NOTE 5.7

Examples of character type declaration statements are:

```
CHARACTER (LEN = 10, KIND = 2)  A
CHARACTER B, C *20
```

5.1.1.6 LOGICAL

The LOGICAL type specifier is used to declare entities of intrinsic type logical (4.3.2.2).

The kind selector, if present, specifies the representation method. If the kind selector is absent, the kind type parameter is KIND (.FALSE.) and the entities declared are of type default logical.

5.1.1.7 Derived type

A TYPE type specifier is used to declare entities of the derived type specified by the *type-name*. The components of each such entity are declared to be of the types specified by the corresponding *component-def* statements of the *derived-type-def* (4.4.1). When a data entity is declared explicitly to be of a derived type, the 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.

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**. A scalar entity of numeric sequence type (4.4.1) is a **numeric sequence structure**. A scalar entity of character sequence type (4.4.1) is a **character sequence structure**.

A declaration for a nonsequence derived-type dummy argument shall specify a derived type that is accessed by use association or host association because the same definition shall be used to declare both the actual and dummy arguments to ensure that both are of the same derived type. This restriction does not apply to arguments of sequence type (4.4.2).

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 PARAMETER attribute

The **PARAMETER attribute** specifies entities that are named constants. The *object-name* becomes defined with the value determined from the *initialization-expr* that appears on the right of the equals, in accordance with the rules of intrinsic assignment (7.5.1.4). The appearance of a PARAMETER attribute in a specification requires that the = *initialization-expr* option appear for all objects in the *entity-decl-list*.

Any named constant that appears in the initialization expression shall have been defined previously in the same type declaration statement, defined in a prior PARAMETER statement or type declaration statement using the PARAMETER attribute, or made accessible by use association or host association. A named constant shall not be referenced in any other context unless it has been defined in a prior PARAMETER statement or type declaration statement using the PARAMETER attribute, or made accessible by use association or host association.

A named constant shall not appear within a format specification (10.1.1).

NOTE 5.8

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 ( ))
```

5.1.2.2 Accessibility attribute

The **accessibility attribute** specifies the accessibility of entities.

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

Constraint: An *access-spec* shall appear only in the *specification-part* of a module.

Entities that are declared with a PRIVATE attribute are not accessible outside the module. Entities that are declared with a PUBLIC attribute may be made accessible in other program units by the

USE statement. Entities without an explicitly specified *access-spec* have default accessibility, which is PUBLIC unless the default has been changed by a PRIVATE statement (5.2.3).

NOTE 5.9

An example of an accessibility specification is:

```
REAL, PRIVATE :: X, Y, Z
```

5.1.2.3 INTENT attribute

An **INTENT attribute** specifies the intended use of the dummy argument.

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

Constraint: The INTENT attribute shall not be specified for a dummy argument that is a dummy procedure or a dummy pointer.

Constraint: A dummy argument with the INTENT(IN) attribute, or a subobject of such a dummy argument, shall not appear as

(1) The *variable* of an *assignment-stmt*,

(2) The *pointer-object* of a *pointer-assignment-stmt*,

(3) A DO variable or implied-DO variable,

(4) An *input-item* in a *read-stmt*,

(5) A *variable-name* in a *namelist-stmt* if the *name-list-group-name* appears in a NML= specifier in a *read-stmt*,

(6) An *internal-file-unit* in a *write-stmt*,

(7) An IOSTAT= or SIZE= specifier in an input/output statement,

(8) A definable variable in an INQUIRE statement,

(9) A *stat-variable* or *allocate-object* in an *allocate-stmt* or a *deallocate-stmt*, or

(10) An actual argument in a reference to a procedure with an explicit interface when the associated dummy argument has the INTENT(OUT) or INTENT(INOUT) attribute.

The INTENT (IN) attribute specifies that the dummy argument shall neither be defined nor become undefined during the execution of the procedure.

The INTENT (OUT) attribute specifies that the dummy argument 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 (INOUT) attribute specifies that the dummy argument is intended for use both to receive data from and to return data to the invoking scoping unit. Any actual argument that becomes associated with such a dummy argument shall be definable.

If no INTENT attribute is specified for a dummy argument, its use is subject to the limitations of the associated actual argument (12.4.1.1, 12.4.1.2, 12.4.1.3).

NOTE 5.10

An example of INTENT specification is:

```
SUBROUTINE MOVE (FROM, TO)
   USE PERSON_MODULE
   TYPE (PERSON), INTENT (IN) :: FROM
   TYPE (PERSON), INTENT (OUT) :: TO
```

5.1.2.4 DIMENSION attribute

The **DIMENSION attribute** specifies entities that are arrays. The rank or 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. An *array-spec* in an *entity-decl* specifies either the rank or the rank and shape for a single array and overrides the *array-spec* in the DIMENSION *attr-spec*. To declare 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.

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

Constraint: 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
```

5.1.2.4.1 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.

R514 *explicit-shape-spec* **is** [*lower-bound* :] *upper-bound*

R515 *lower-bound* **is** *specification-expr*

R516 *upper-bound* **is** *specification-expr*

Constraint: An explicit-shape array whose bounds depend on the values of nonconstant 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 nonconstant specification expressions.

If an explicit-shape array has bounds that are nonconstant specification 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 any redefinition or undefinition of the specification expression variables 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.

5.1.2.4.2 Assumed-shape array

An **assumed-shape array** is a nonpointer dummy argument array that takes its shape from the associated actual argument array.

R517 *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 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.

5.1.2.4.3 Deferred-shape array

A **deferred-shape array** is an allocatable array or an array pointer.

An **allocatable array** is a named array that has the ALLOCATABLE attribute and a specified rank, but its bounds, and hence shape, are determined when space is allocated for the array by execution of an ALLOCATE statement (6.3.1).

The ALLOCATABLE attribute may be specified for an array in a type declaration statement or in an ALLOCATABLE statement (5.2.6). An array with the ALLOCATABLE attribute shall be declared with a *deferred-shape-spec-list* in a type declaration statement, an ALLOCATABLE statement, a DIMENSION statement (5.2.5), or a TARGET statement (5.2.8). The type and type parameters may be specified in a type declaration statement.

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 by pointer assignment (7.5.2) or by execution of an ALLOCATE statement (6.3.1). The POINTER attribute may be specified for an array in a type declaration statement, a component definition statement, or a POINTER statement (5.2.7). An array with the POINTER attribute shall be declared with a *deferred-shape-spec-list* in a type declaration statement, a POINTER statement, or a DIMENSION statement (5.2.5). The type and type parameters may be specified in a type declaration statement or a component definition statement.

R518 *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 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 that is inquiring about the allocation status or a property of the type or type parameters. The lower and upper bounds of each dimension are those specified in the ALLOCATE statement when the array is allocated.

The size, bounds, and shape of the target of 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 that is inquiring about argument presence, a property of the type or type parameters, or association status. The bounds of each dimension of an array pointer may be specified in two ways:

(1) They are specified in an ALLOCATE statement (6.3.1) when the target is allocated, or

(2) They are specified in a pointer assignment statement. The lower bound of each dimension is the result of the intrinsic function LBOUND (13.14.53) applied to the corresponding dimension of the target. The upper bound of each dimension is the result of the intrinsic function UBOUND (13.14.113) applied to the corresponding dimension of the target.

The bounds of the array target or allocatable array are unaffected by any subsequent redefinition or undefinition of variables involved in the bounds.

5.1.2.4.4 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.

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

Constraint: The function name of an array-valued function shall not be declared as an assumed-size array.

Constraint: 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.

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 or in a reference to the intrinsic function LBOUND.

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 nonconstant specification expressions, the bounds are declared at entry to the procedure. The bounds of such an array are unaffected by any redefinition or undefinition of the specification expression variables during execution of the procedure.

5.1.2.5 SAVE attribute

An object with the **SAVE attribute**, declared 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 the object is a pointer and its target becomes undefined (14.6.2.1.3(3)). The object is shared by all instances (12.5.2.4) of the subprogram. Such an object is called a **saved object**.

An object 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 the object is a pointer and its target becomes undefined.

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

5.1.2.6 OPTIONAL attribute

The **OPTIONAL attribute** shall be specified only in the scoping unit of a subprogram or an interface block, and shall be specified only for dummy arguments. The OPTIONAL attribute specifies that the dummy argument need not be associated with an actual argument in a reference to the procedure (12.4.1.5). The PRESENT intrinsic function may be used to determine whether an actual argument has been associated with a dummy argument having the OPTIONAL attribute.

5.1.2.7 POINTER attribute

An object with the **POINTER attribute** shall neither be referenced nor defined unless, as a result of executing a pointer assignment (7.5.2) or an ALLOCATE statement (6.3.1), it becomes pointer associated with a target object that may be referenced or defined. If the pointer is an array, it shall be declared with a *deferred-shape-spec-list*.

NOTE 5.12

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.

5.1.2.8 TARGET attribute

An object with the **TARGET attribute** may have a pointer associated with it (7.5.2). An object without the TARGET or POINTER attribute shall not have an accessible pointer associated with it.

NOTE 5.13

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.

5.1.2.9 ALLOCATABLE attribute

The **ALLOCATABLE attribute** may be specified for an array. Such an array shall be a deferred-shape array; the shape is determined when space is allocated for the array by execution of an ALLOCATE statement (6.3.1).

5.1.2.10 EXTERNAL attribute

The **EXTERNAL attribute** specifies an external function or a dummy function and permits the function name to be used as an actual argument. This attribute may also be declared via the EXTERNAL statement (12.3.2.2).

5.1.2.11 INTRINSIC attribute

The **INTRINSIC attribute** specifies the specific or generic name of an intrinsic function and permits the name to be used as an actual argument if it is a specific name of an intrinsic function (13.13). This attribute may also be declared via the INTRINSIC statement (12.3.2.3).

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 EXTERNAL and INTRINSIC statements.

5.2.1 INTENT statement

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

Constraint: An *intent-stmt* shall appear only in the *specification-part* of a subprogram or an interface body (12.3.2.1).

Constraint: *dummy-arg-name* shall not be the name of a dummy procedure or a dummy pointer.

This statement specifies the INTENT attribute (5.1.2.3) for the dummy arguments in the list.

NOTE 5.14

An example of an INTENT statement is:

```
SUBROUTINE EX (A, B)
   INTENT (INOUT) :: A, B
```

5.2.2 OPTIONAL statement

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

Constraint: An *optional-stmt* shall occur only in the *specification-part* of a subprogram or an interface body (12.3.2.1).

This statement specifies the OPTIONAL attribute (5.1.2.6) for the dummy arguments in the list.

NOTE 5.15

An example of an OPTIONAL statement is:

```
SUBROUTINE EX (A, B)
   OPTIONAL :: B
```

5.2.3 Accessibility statements

R522 *access-stmt* **is** *access-spec* [[::] *access-id-list*]

R523 *access-id* **is** *use-name*
or *generic-spec*

Constraint: 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.

Constraint: Each *use-name* shall be the name of a named variable, procedure, derived type, named constant, or namelist group.

Constraint: A module procedure that has a dummy argument or function result of a type that has PRIVATE accessibility shall have PRIVATE accessibility and shall not have a generic identifier that has PUBLIC accessibility.

An *access-stmt* with an *access-id-list* specifies the accessibility attribute (5.1.2.2), PUBLIC or PRIVATE, of the entities in the list. An *access-stmt* without an *access-id* list specifies the default accessibility that applies to all potentially accessible entities 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 5.16

Examples of accessibility statements are:

```
MODULE EX
   PRIVATE
   PUBLIC :: A, B, C, ASSIGNMENT (=), OPERATOR (+)
```

5.2.4 SAVE statement

R524 *save-stmt* **is** SAVE [[::] *saved-entity-list*]

R525 *saved-entity* **is** *object-name*
or / *common-block-name* /

Constraint: An *object-name* shall not be the name of an object in a common block, a dummy argument name, a procedure name, a function result name, an automatic data object name, or the name of an object with the PARAMETER attribute.

Constraint: 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 (5.1.2.5) for all objects 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 5.17

An example of a SAVE statement is:

```
SAVE A, B, C, / BLOCKA /, D
```

5.2.5 DIMENSION statement

R526 *dimension-stmt* **is** DIMENSION [::] *array-name* (*array-spec*) ■
■ [, *array-name* (*array-spec*)] ...

This statement specifies the DIMENSION attribute (5.1.2.4) and the array properties for each object named.

NOTE 5.18

An example of a DIMENSION statement is:

```
DIMENSION A (10), B (10, 70), C (:)
```

5.2.6 ALLOCATABLE statement

R527 *allocatable-stmt* **is** ALLOCATABLE [::] ■
■ *array-name* [(*deferred-shape-spec-list*)] ■
■ [, *array-name* [(*deferred-shape-spec-list*)]] ...

Constraint: The *array-name* shall not be a dummy argument or function result.

Constraint: If the DIMENSION attribute for an *array-name* is specified elsewhere in the scoping unit, the *array-spec* shall be a *deferred-shape-spec-list*.

This statement specifies the ALLOCATABLE attribute (5.1.2.9) for a list of arrays.

NOTE 5.19

An example of an ALLOCATABLE statement is:

```
REAL A, B (:)
ALLOCATABLE :: A (:, :), B
```

5.2.7 POINTER statement

R528 *pointer-stmt* **is** POINTER [::] *object-name* [(*deferred-shape-spec-list*)] ■
■ [, *object-name* [(*deferred-shape-spec-list*)]] ...

Constraint: The INTENT attribute shall not be specified for an *object-name*.

Constraint: If the DIMENSION attribute for an *object-name* is specified elsewhere in the scoping unit, the *array-spec* shall be a *deferred-shape-spec-list*.

Constraint: The PARAMETER attribute shall not be specified for an *object-name*.

This statement specifies the POINTER attribute (5.1.2.7) for a list of objects.

NOTE 5.20

An example of a POINTER statement is:

```
TYPE (NODE) :: CURRENT
POINTER :: CURRENT, A (:, :)
```

5.2.8 TARGET statement

R529 *target-stmt* **is** TARGET [::] *object-name* [(*array-spec*)] ■
■ [, *object-name* [(*array-spec*)]] ...

Constraint: The PARAMETER attribute shall not be specified for an *object-name*.

This statement specifies the TARGET attribute (5.1.2.8) for a list of objects.

NOTE 5.21

An example of a TARGET statement is:

```
TARGET :: A (1000, 1000), B
```

5.2.9 PARAMETER statement

The **PARAMETER statement** specifies the PARAMETER attribute (5.1.2.1) and the values for the named constants in the list.

R530 *parameter-stmt* **is** PARAMETER (*named-constant-def-list*)

R531 *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.

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

A named constant that appears in the initialization expression shall have been defined previously in the same PARAMETER statement, defined in a prior PARAMETER statement or type declaration statement using the PARAMETER attribute, or made accessible by use association or host association.

A named constant shall not appear as part of a format specification (10.1.1).

NOTE 5.22

An example of a PARAMETER statement is:

```
PARAMETER (MODULUS = MOD (28, 3), NUMBER_OF_SENATORS  = 100)
```

5.2.10 DATA statement

R532 *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 or subobject 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.

R533 *data-stmt-set* **is** *data-stmt-object-list* / *data-stmt-value-list* /

R534 *data-stmt-object* **is** *variable*
or *data-implied-do*

R535 *data-implied-do* **is** (*data-i-do-object-list* , *data-i-do-variable* = ■
■ *scalar-int-expr* , *scalar-int-expr* [, *scalar-int-expr*])

R536 *data-i-do-object* **is** *array-element*
or *scalar-structure-component*
or *data-implied-do*

R537 *data-i-do-variable* **is** *scalar-int-variable*

Constraint: 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.

Constraint: A variable whose name or 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 array.

Constraint: A *data-i-do-object* or a *variable* that appears as a *data-stmt-object* shall not be a subobject of a pointer.

Constraint: *data-i-do-variable* shall be a named variable.

Constraint: 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.

Constraint: The *array-element* shall not have a constant parent.

Constraint: The *scalar-structure-component* shall not have a constant parent.

Constraint: The *scalar-structure-component* shall contain at least one *part-ref* that contains a *subscript-list*.

Constraint: 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 the containing *data-implied-do*s, and each operation shall be intrinsic.

R538 *data-stmt-value* **is** [*data-stmt-repeat* *] *data-stmt-constant*

R539 *data-stmt-repeat* **is** *scalar-int-constant*
or *scalar-int-constant-subobject*

R540 *data-stmt-constant* **is** *scalar-constant*
or *scalar-constant-subobject*
or *signed-int-literal-constant*
or *signed-real-literal-constant*
or NULL ( )
or *structure-constructor*

Constraint: 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.

Constraint: In a *scalar-int-constant-subobject* that is a *data-stmt-repeat* any subscript shall be an initialization expression.

Constraint: In a *scalar-constant-subobject* that is a *data-stmt-constant* any subscript, substring starting point, or substring ending point shall be an initialization expression.

Constraint: 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.

Constraint: If a *data-stmt-constant* is a *structure-constructor*, each component 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 implied-DO variable, as in the DO construct (8.1.4.4).

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 an implied-DO list 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 list. 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 ( ) for the corresponding variable. The lengths of the two expanded sequences shall be the same.

If a *data-stmt-constant* is NULL ( ), the corresponding *data-stmt-object* shall have the POINTER attribute.

If a *data-statement-constant* is a *boz-literal-constant*, the corresponding object shall be of type integer. A *data-stmt-constant* that is a *boz-literal-constant* is treated as if the constant were an *int-literal-constant* with a *kind-param* that specifies the representation method with the largest decimal exponent range supported by the processor.

A *data-stmt-constant* other than NULL ( ) shall be compatible with its corresponding variable according to the rules of intrinsic assignment (7.5.1.4), and the variable becomes initially defined with the *data-stmt-constant* in accordance with the rules of intrinsic assignment.

If a variable has the POINTER attribute, the corresponding *data-stmt-constant* shall be NULL ( ), and the pointer has an initial association status of disassociated.

NOTE 5.23

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 4.20. The pointer HEAD_OF_LIST is declared using the derived type NODE from Note 4.26; it is initially disassociated. MYNAME is initialized by a structure constructor. YOURNAME is initialized by supplying a separate value for each component.

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.

R541 *implicit-stmt* **is** IMPLICIT *implicit-spec-list*
or IMPLICIT NONE

R542 *implicit-spec* **is** *type-spec* (*letter-spec-list*)

R543 *letter-spec* **is** *letter* [– *letter*]

Constraint: 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.

Constraint: 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 5.24

The following are examples of the use of IMPLICIT statements:

```
MODULE EXAMPLE_MODULE
   IMPLICIT NONE
   ...
   INTERFACE
      FUNCTION FUN (I)   ! Not all data entities need
         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
      ...
   END SUBROUTINE SUB2
```

NOTE 5.24 *(Continued)*

```
   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 5.25

An IMPLICIT statement may specify a *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 5.26

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 (9.4, 10.9).

R544 *namelist-stmt* **is** NAMELIST ■
■ / *namelist-group-name* / *namelist-group-object-list* ■
■ [[,] / *namelist-group-name* / *namelist-group-object-list*] ...

R545 *namelist-group-object* **is** *variable-name*

Constraint: A *namelist-group-object* shall not be an array dummy argument with a nonconstant bound, a variable with nonconstant character length, an automatic object, a pointer, a variable of a type that has an ultimate component that is a pointer, or an allocatable array.

Constraint: If a *namelist-group-name* has the PUBLIC attribute, no item in the *namelist-group-object-list* shall have the PRIVATE attribute or have private components.

Constraint: The *namelist-group-name* shall not be a name made accessible by use association.

The order in which the data objects (variables) are specified in the NAMELIST statement determines the order in which the values appear on output.

Any *namelist-group-name* may occur in more than one NAMELIST statement 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 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 5.27

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 statement, the COMMON statement, and the SEQUENCE statement provide for control of the order and layout of storage units. The general mechanism of storage association is described in 14.6.3.

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.

R546 *equivalence-stmt* **is** EQUIVALENCE *equivalence-set-list*

R547 *equivalence-set* **is** (*equivalence-object* , *equivalence-object-list*)

R548 *equivalence-object* **is** *variable-name*
or *array-element*
or *substring*

Constraint: An *equivalence-object* shall not be a dummy argument, a pointer, an allocatable array, an object of a nonsequence derived type, an object of a sequence derived type containing a pointer at any level of component selection, an automatic object, a function name, an entry name, a result name, a named constant, a structure component, or a subobject of any of the preceding objects.

Constraint: An *equivalence-object* shall not have the TARGET attribute.

Constraint: Each subscript or substring range expression in an *equivalence-object* shall be an integer initialization expression (7.1.6.1).

Constraint: 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.

Constraint: 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.

Constraint: If an *equivalence-object* is of a 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.

Constraint: 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.

Constraint: The name of an *equivalence-object* shall not be a name made accessible by use association.

Constraint: A *substring* shall not have length zero.

NOTE 5.28

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.

Structures that appear in EQUIVALENCE statements shall be sequence structures. 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.

A numeric sequence structure may be equivalenced to another numeric sequence structure, 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 character sequence structure may be equivalenced to an object of default character type or another character sequence structure.

Other objects 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 14.6.3.3.

5.5.1.1 Equivalence association

An EQUIVALENCE statement specifies that the storage sequences (14.6.3.1) 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 5.29

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 5.30

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 5.31

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 (14.6.3).

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**.

R549 *common-stmt* **is** COMMON ■
■ [/ [*common-block-name*] /]*common-block-object-list* ■
■ [[,] / [*common-block-name*] /*common-block-object-list*] ...

R550 *common-block-object* **is** *variable-name* [(*explicit-shape-spec-list*)]

Constraint: Only one appearance of a given *variable-name* is permitted in all *common-block-object-list*s within a scoping unit.

Constraint: A *common-block-object* shall not be a dummy argument, an allocatable array, an automatic object, a function name, an entry name, or a result name.

Constraint: Each bound in the *explicit-shape-spec* shall be a constant specification expression (7.1.6.2).

Constraint: If a *common-block-object* is of a derived type, it shall be a sequence type (4.4.1) with no default initialization.

Constraint: If a *variable-name* appears with an *explicit-shape-spec-list*, it shall not have the POINTER attribute.

Constraint: A *variable-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 (4.4.1) or character sequence type (4.4.1) appear in common, it is as if the individual components were enumerated directly in the common list.

NOTE 5.32

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 (14.6.3.1) 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. Variables made accessible by use association or host association do not contribute.

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 become associated only with nonpointer objects of these types.

A nonpointer object of type default character or character sequence type shall become 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 become associated only with nonpointer objects of the same type.

A nonpointer object of intrinsic type other than default integer, default real, double precision real, default complex, default logical, or default character shall become associated only with nonpointer objects of the same type and type parameters.

A pointer shall become storage associated only with pointers of the same type, type parameters, and rank.

An object with the TARGET attribute may become storage associated only with another object that has the TARGET attribute and the same type and type parameters.

NOTE 5.33

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 (14.7.6).

(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 (11.4), 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 5.34

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 name or subobject 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 the data object name or subobject designator appears in certain contexts and when certain events occur (14.7).

R601 *variable* **is** *scalar-variable-name*
or *array-variable-name*
or *subobject*

Constraint: *array-variable-name* shall be the name of a data object that is an array.

Constraint: *array-variable-name* shall not have the PARAMETER attribute.

Constraint: *scalar-variable-name* shall not have the PARAMETER attribute.

Constraint: *subobject* shall not be a subobject designator (for example, a substring) whose parent is a constant.

R602 *subobject* **is** *array-element*
or *array-section*
or *structure-component*
or *substring*

R603 *logical-variable* **is** *variable*

Constraint: *logical-variable* shall be of type logical.

R604 *default-logical-variable* **is** *variable*

Constraint: *default-logical-variable* shall be of type default logical.

R605 *char-variable* **is** *variable*

Constraint: *char-variable* shall be of type character.

R606 *default-char-variable* **is** *variable*

Constraint: *default-char-variable* shall be of type default character.

R607 *int-variable* **is** *variable*

Constraint: *int-variable* shall be of type integer.

R608 *default-int-variable* **is** *variable*

Constraint: *default-int-variable* shall be of type default integer.

Pointers and allocatable arrays shall not be defined in circumstances explained in 5.1.2.4.3. Dummy arguments or variables associated with dummy arguments shall not be defined in circumstances explained in 12.4.1.1, 12.4.1.5, 12.4.1.6, and 12.5.2.1.

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 been associated with a name with the PARAMETER attribute (5.1.2.1, 5.2.9). A reference to a constant is always permitted; redefinition of a constant is never permitted.

NOTE 6.1

For example, given the declarations:

```
CHARACTER (10)  A, B (10)
TYPE (PERSON)  P   ! See Note 4.20
```

then A, B, B (1), B (1:5), P % AGE, and A (1:1) are all variables.

6.1 Scalars

A **scalar** (2.4.4) is a data entity that can be represented by a single value of the data 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 data type.

NOTE 6.2

A scalar object of derived type has a single value that consists of values of the data types of its components (4.4.3).

A scalar has rank zero.

6.1.1 Substrings

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

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*]

Constraint: *parent-string* shall be of type character.

The first *scalar-int-expr* in *substring-range* is called the **starting point** and the second one is called the **ending point**. The length of a substring is the number of characters in the substring and is MAX ($l - f + 1$, 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 a subobject 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*)]

Constraint: In a *data-ref*, each *part-name* except the rightmost shall be of derived type.

Constraint: In a *data-ref*, each *part-name* except the leftmost shall be the name of a component of the derived-type definition of the type of the preceding *part-name*.

Constraint: In a *part-ref* containing a *section-subscript-list*, 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.

Constraint: In a *data-ref*, 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 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 **parent object** of a *data-ref* is the data object whose name is the leftmost part name.

R614 *structure-component* **is** *data-ref*

Constraint: In a *structure-component*, there shall be more than one *part-ref* and the rightmost *part-ref* shall be of the form *part-name*.

The type and type parameters, if any, of a structure component are those of the rightmost part name. A structure component shall be neither referenced nor defined before the declaration of the parent object. A structure component has the INTENT, TARGET, or PARAMETER attribute if the parent object has the attribute. A structure component is a pointer only if the rightmost part name is defined to have the POINTER attribute.

NOTE 6.4

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 6.5

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 non-zero 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.

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 name or 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 (5.1.2.1, 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

R615 *array-element* **is** *data-ref*

Constraint: In an *array-element*, every *part-ref* shall have rank zero and the last *part-ref* shall contain a *subscript-list*.

R616 *array-section* **is** *data-ref* [(*substring-range*)]

Constraint: In an *array-section*, 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.

Constraint: In an *array-section* with a *substring-range*, the rightmost *part-name* shall be of type character.

R617 *subscript* **is** *scalar-int-expr*

R618 *section-subscript* **is** *subscript*
or *subscript-triplet*
or *vector-subscript*

R619 *subscript-triplet* **is** [*subscript*] : [*subscript*] [: *stride*]

R620 *stride* **is** *scalar-int-expr*

R621 *vector-subscript* **is** *int-expr*

Constraint: A *vector-subscript* shall be an integer array expression of rank one.

Constraint: 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 6.6

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.

An array element or an array section has the INTENT, TARGET, or PARAMETER attribute if its parent has the attribute, but it never has the POINTER attribute.

NOTE 6.7

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 i th 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 *i*th element is the number of integer values in the sequence indicated by the *i*th 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 second subscript shall not be omitted in the last dimension of an assumed-size array.

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

The stride shall not be zero.

NOTE 6.8

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 6.9

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 6.10

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 6.11

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 creation, 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 (7.5.2) 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 arrays with the ALLOCATABLE attribute.

NOTE 6.12

For detailed remarks regarding pointers and dynamic association see C.3.2.

6.3.1 ALLOCATE statement

The **ALLOCATE statement** dynamically creates pointer targets and allocatable arrays.

R622 *allocate-stmt* **is** ALLOCATE (*allocation-list* [, STAT = *stat-variable*])

R623 *stat-variable* **is** *scalar-int-variable*

R624 *allocation* **is** *allocate-object* [(*allocate-shape-spec-list*)]

R625 *allocate-object* **is** *variable-name*
or *structure-component*

R626 *allocate-shape-spec* **is** [*allocate-lower-bound* :] *allocate-upper-bound*

R627 *allocate-lower-bound* **is** *scalar-int-expr*

R628 *allocate-upper-bound* **is** *scalar-int-expr*

Constraint: Each *allocate-object* shall be a pointer or an allocatable array.

Constraint: The number of *allocate-shape-spec*s in an *allocate-shape-spec-list* shall be the same as the rank of the pointer or allocatable array.

An *allocate-object*, or a subobject of an *allocate-object*, shall not appear in a bound in the same ALLOCATE statement. The *stat-variable* shall not appear in a bound in the same ALLOCATE statement.

NOTE 6.13

An example of an ALLOCATE statement is:

```
ALLOCATE (X (N), B (-3 : M, 0:9), STAT = IERR_ALLOC)
```

The *stat-variable* shall not be allocated within the ALLOCATE statement in which it appears; nor shall it depend on the value, bounds, allocation status, or association status of any *allocate-object* or subobject of an *allocate-object* allocated in the same statement.

At the time 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 6.14

An *allocate-object* may be of type character with zero character length.

If the STAT= specifier is present, 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 be currently allocated or be 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.

6.3.1.1 Allocation of allocatable arrays

An allocatable array that has been allocated by an ALLOCATE statement and has not been subsequently deallocated (6.3.3) is **currently allocated** and is definable. Allocating a currently allocated allocatable array causes an error condition in the ALLOCATE statement. At the beginning of execution of a program, allocatable arrays have the allocation status of not currently allocated and are not definable. The ALLOCATED intrinsic function (13.14.9) may be used to determine whether an allocatable array is currently allocated.

6.3.1.2 Allocation status

The allocation status of an allocatable array is one of the following at any time during the execution of a program:

(1) Not currently allocated. An allocatable array with this status shall not be referenced or defined; it may be allocated with the ALLOCATE statement. Deallocating it causes an error condition in the DEALLOCATE statement. The ALLOCATED intrinsic returns .FALSE. for such an array.

(2) Currently allocated. An allocatable array with this status may be referenced, defined, or deallocated; allocating it causes an error condition in the ALLOCATE statement. The ALLOCATED intrinsic returns .TRUE. for such an array.

An allocatable array with the SAVE attribute has an initial status of not currently allocated. If the array is allocated, its status changes to currently allocated. The status remains currently allocated until the array is deallocated.

An allocatable array that does not have the SAVE attribute, that is a local variable of a procedure, and that is not accessed by use or host association, has a status of not currently allocated at the beginning of each invocation of the procedure. The status may change during execution of the

procedure. If the array has a status of currently allocated when the procedure is exited by execution of a RETURN or END statement, it is deallocated.

An allocatable array that does not have the SAVE attribute and that is accessed by use association has an initial status of not currently allocated. The status may change during execution of the program. If the array has an allocation status of currently allocated when execution of a RETURN or END statement results in no scoping unit having access to the module, it is processor dependent whether the allocation status remains currently allocated or the array is deallocated.

NOTE 6.15

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 not saved 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))
   IF (.NOT. ALLOCATED(UNSAVED2)) ALLOCATE(UNSAVED2(10))
   END SUBROUTINE SUB1
END PROGRAM MAIN
```

6.3.1.3 Allocation of pointer targets

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. Allocation of a pointer creates an object that implicitly has the TARGET attribute. 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 currently 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 specified in 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 it can still be referred to by other pointers that are currently associated with it. The ASSOCIATED intrinsic function (13.14.13) may be used to determine whether a pointer is currently 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.

6.3.2 NULLIFY statement

The **NULLIFY statement** causes pointers to be disassociated.

R629 *nullify-stmt* **is** NULLIFY (*pointer-object-list*)

R630 *pointer-object* **is** *variable-name*
or *structure-component*

Constraint: 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*, or subobject of another *pointer-object*, in the same NULLIFY statement.

6.3.3 DEALLOCATE statement

The **DEALLOCATE statement** causes allocatable arrays to be deallocated and it causes pointer targets to be deallocated and the pointers to be disassociated.

R631 *deallocate-stmt* **is** DEALLOCATE (*allocate-object-list* [, STAT = *stat-variable*])

Constraint: Each *allocate-object* shall be a pointer or an allocatable array.

An *allocate-object* shall not depend on the value, bounds, allocation status, or association status of another *allocate-object* or subobject of another allocate-object in the same DEALLOCATE statement; nor shall it depend on the value of the *stat-variable* in the same DEALLOCATE statement.

The *stat-variable* shall not be deallocated within the same DEALLOCATE statement; nor shall it depend on the value, bounds, allocation status, or association status of any *allocate-object* or subobject of an *allocate-object* in the same DEALLOCATE statement.

If the STAT= specifier is present, successful execution of the DEALLOCATE statement causes the *stat-variable* to become defined with a value of zero. If an error condition occurs during the execution of the DEALLOCATE statement, the *stat-variable* becomes defined with a processor-dependent positive integer value and each *allocate-object* that was successfully deallocated shall be not currently allocated or shall be disassociated. Each *allocate-object* that was not successfully deallocated shall retain its previous allocation status or pointer association status.

NOTE 6.16

The status of objects that were not successfully deallocated can be individually checked with the ALLOCATED or ASSOCIATED intrinsic functions.

If an error condition occurs during execution of a DEALLOCATE statement that does not contain the STAT= specifier, execution of the program is terminated.

NOTE 6.17

An example of a DEALLOCATE statement is:

```
DEALLOCATE (X, B)
```

6.3.3.1 Deallocation of allocatable arrays

Deallocating an allocatable array that is not currently allocated causes an error condition in the DEALLOCATE statement. An allocatable array with the TARGET attribute shall not be deallocated through an associated pointer. Deallocating an allocatable array 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 array that is a local variable of the procedure retains its allocation and definition status if

(1) It has the SAVE attribute,

(2) It is accessed by use association, if the module defining the array is also accessed by another scoping unit that is currently in execution, or

(3) It is accessed by host association.

When the execution of a procedure is terminated by execution of a RETURN or END statement, an allocatable array that is a local variable of the procedure and is not included in the above categories has allocation status as follows:

(1) If it is accessed by use association, its allocation status is processor dependent.

(2) Otherwise, it is deallocated (as if by a DEALLOCATE statement).

NOTE 6.18

The ALLOCATED intrinsic function may be used to determine whether an array is still currently allocated or has been deallocated.

NOTE 6.19

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 not currently allocated.

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 currently associated with an allocatable array, the pointer shall not be deallocated.

A pointer that is not currently associated with the whole of an allocated target object shall not be deallocated. If a pointer is currently associated with a portion (2.4.3.1) of a target object that is independent of any other portion of the target object, it shall not be deallocated. 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.

When the execution of a procedure is terminated by execution of a RETURN or END statement, the pointer association status of a pointer declared or accessed in the subprogram that defines the procedure becomes undefined unless it is one of the following:

(1) A pointer with the SAVE attribute,

(2) A pointer in blank common,

(3) A pointer in a named common block that appears in at least one other scoping unit that is currently in execution,

(4) A pointer declared in the scoping unit of a module if the module also is accessed by another scoping unit that is currently in execution,

(5) A pointer accessed by host association, or

(6) A pointer that is the return value of a function declared to have the POINTER attribute.

When a pointer target becomes undefined by execution of a RETURN or END statement, the pointer association status (14.6.2.1) becomes undefined.

Section 7: Expressions and assignment

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

7.1 Expressions

An **expression** represents 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.

NOTE 7.1

Simple forms of an operand are constants and variables, such as:

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

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

NOTE 7.2

Examples of intrinsic operators are:

```
+
*
>
.AND.
```

7.1.1 Form of an expression

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

NOTE 7.3

Examples of expressions are:

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

An expression is defined in terms of several categories: primary, level-1 expression, level-2 expression, level-3 expression, level-4 expression, 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 and 7.3.

7.1.1.1 Primary

R701 *primary* **is** *constant*
or *constant-subobject*
or *variable*
or *array-constructor*
or *structure-constructor*
or *function-reference*
or (*expr*)

R702 *constant-subobject* **is** *subobject*

Constraint: *subobject* shall be a subobject designator whose parent is a constant.
A variable that is a primary shall not be a whole assumed-size array.

NOTE 7.4

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*
`(S + T)`	(*expr*)

7.1.1.2 Level-1 expressions

Defined unary operators have the highest operator precedence (Table 7.8). Level-1 expressions are primaries optionally operated on by defined unary operators:

R703 *level-1-expr* **is** [*defined-unary-op*] *primary*

R704 *defined-unary-op* **is** **.** *letter* [*letter*] ... **.**

Constraint: A *defined-unary-op* shall not contain more than 31 letters and shall not be the same as any *intrinsic-operator* or *logical-literal-constant*.

NOTE 7.5

Simple examples of a level-1 expression are:

Example	Syntactic class
`A`	*primary* (R701)
`.INVERSE. B`	*level-1-expr* (R703)

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

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

7.1.1.3 Level-2 expressions

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

R705 *mult-operand* **is** *level-1-expr* [*power-op mult-operand*]

R706 *add-operand* **is** [*add-operand mult-op*] *mult-operand*

R707 *level-2-expr* **is** [[*level-2-expr*] *add-op*] *add-operand*

R708 *power-op* **is** **

R709 *mult-op* **is** *
or /

R710 *add-op* **is** +
or –

NOTE 7.6

Simple examples of a level-2 expression are:

Example	Syntactic class	Remarks
`A`	*level-1-expr*	A is a *primary*. (R703)
`B ** C`	*mult-operand*	B is a *level-1-expr*, ** is a *power-op*, and C is a *mult-operand*. (R705)
`D * E`	*add-operand*	D is an *add-operand*, * is a *mult-op*, and E is a *mult-operand*. (R706)
`+1`	*level-2-expr*	+ is an *add-op* and 1 is an *add-operand*. (R707)
`F - I`	*level-2-expr*	F is a *level-2-expr*, – is an *add-op*, and I is an *add-operand*. (R707)

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

```
- A + D * E + B ** C
```

7.1.1.4 Level-3 expressions

Level-3 expressions are level-2 expressions optionally involving the character operator *concat-op*.

R711 *level-3-expr* **is** [*level-3-expr concat-op*] *level-2-expr*

R712 *concat-op* **is** //

NOTE 7.7

Simple examples of a level-3 expression are:

Example	Syntactic class
`A`	*level-2-expr* (R707)
`B // C`	*level-3-expr* (R711)

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*.

R713 *level-4-expr* **is** [*level-3-expr rel-op*] *level-3-expr*

R714 *rel-op* **is** **.**EQ**.**
or **.**NE**.**
or **.**LT**.**
or **.**LE**.**
or **.**GT**.**
or **.**GE**.**
or ==
or /=
or <
or <=
or >

or >=

NOTE 7.8

Simple examples of a level-4 expression are:

Example	Syntactic class
`A`	*level-3-expr* (R711)
`B .EQ. C`	*level-4-expr* (R713)
`D < E`	*level-4-expr* (R713)

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

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

R715 *and-operand* **is** [*not-op*] *level-4-expr*

R716 *or-operand* **is** [*or-operand and-op*] *and-operand*

R717 *equiv-operand* **is** [*equiv-operand or-op*] *or-operand*

R718 *level-5-expr* **is** [*level-5-expr equiv-op*] *equiv-operand*

R719 *not-op* **is** **.**NOT**.**

R720 *and-op* **is** **.**AND**.**

R721 *or-op* **is** **.**OR**.**

R722 *equiv-op* **is** **.**EQV**.**
or **.**NEQV**.**

NOTE 7.9

Simple examples of a level-5 expression are:

Example	Syntactic class
`A`	*level-4-expr* (R713)
`.NOT. B`	*and-operand* (R715)
`C .AND. D`	*or-operand* (R716)
`E .OR. F`	*equiv-operand* (R717)
`G .EQV. H`	*level-5-expr* (R718)
`S .NEQV. T`	*level-5-expr* (R718)

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 7.8).

R723 *expr* **is** [*expr defined-binary-op*] *level-5-expr*

R724 *defined-binary-op* **is** **.** *letter* [*letter*] ... **.**

Constraint: A *defined-binary-op* shall not contain more than 31 letters and shall not be the same as any *intrinsic-operator* or *logical-literal-constant*.

NOTE 7.10

Simple examples of an expression are:

Example	Syntactic class
`A`	*level-5-expr* (R718)
`B.UNION.C`	*expr* (R723)

More complicated examples of an expression are:

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

7.1.2 Intrinsic operations

An **intrinsic operation** is either an intrinsic unary operation or an intrinsic binary operation. An **intrinsic unary operation** is an operation of the form *intrinsic-operator* x_2 where x_2 is of an intrinsic type (4.3) 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 (4.3) 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 requiring 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.

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 and have the same kind type parameter value.

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 and that is defined by a function and a generic interface block (12.3.2.1) or that has the form *intrinsic-operator* x_2 where the type of x_2 is not that required for the unary intrinsic operation (7.1.2), and that is defined by a function and a generic interface block.

A **defined binary operation** is an operation that has the form x_1 *defined-binary-op* x_2 and that is defined by a function and a generic interface block or that has the form x_1 *intrinsic-operator* x_2 where the types or ranks of either x_1 or x_2 or both are not those required for the intrinsic binary operation (7.1.2), and that is defined by a function and a generic interface block.

NOTE 7.11

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 Data type, type parameters, and shape of an expression

The data type and shape of an expression depend on the operators and on the data types and shapes of the primaries used in the expression, and are determined recursively from the syntactic form of the expression. The data type of an expression is one of the intrinsic types (4.3) or a derived type (4.4).

R725 *logical-expr* **is** *expr*

Constraint: *logical-expr* shall be of type logical.

R726 *char-expr* **is** *expr*

Constraint: *char-expr* shall be of type character.

R727 *default-char-expr* **is** *expr*

Constraint: *default-char-expr* shall be of type default character.

R728 *int-expr* **is** *expr*

Constraint: *int-expr* shall be of type integer.

R729 *numeric-expr* **is** *expr*

Constraint: *numeric-expr* shall be of type integer, real or complex.

An expression whose type is intrinsic has a kind type parameter. In addition, an expression of type character has a character length parameter. The type parameters for an expression are determined from the form of the expression.

7.1.4.1 Data type, type parameters, and shape of a primary

The data type, type parameters, and shape of a primary are determined according to whether the primary is a constant, variable, array constructor, structure constructor, function reference, 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 is determined by the constructor name. If it is an array constructor, its type, type parameters, and shape are as described in 4.5. 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. In the case of a function reference, the function may be generic (12.3.2.1, 13.11), in which case its type, type parameters, and shape are determined by the types, type parameters, and ranks of its actual arguments.

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) As the *expr* of a parenthesized primary, or

(3) As 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.

The intrinsic function NULL (13.14.79) returns a disassociated pointer. A disassociated pointer has no shape but does have rank. The data type, type parameters, and rank of the result of the intrinsic function NULL when it appears without an argument are determined by the pointer that becomes associated with the result. See Table 7.2.

Table 7.2 Type, type parameters, and rank 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

The optional argument of the intrinsic function NULL shall be present when a reference to the intrinsic function appears as an actual argument in a reference to a generic procedure if the type, type parameters, or rank is required to resolve the generic reference.

NOTE 7.12

For example:

```
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
```

7.1.4.2 Data 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 type of the result of a defined operation $[x_1]\ op\ x_2$ is specified by the function defining the operation (7.3).

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.

An expression of an intrinsic type has a kind type parameter. An expression of type character also has a character length parameter. For an expression x_1 // x_2 where 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. 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. 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. 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 or is processor dependent if the operands have the same decimal exponent range. 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 or is processor dependent if the operands have the same decimal precision. 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. 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 Scalar and array expressions

An expression is either a scalar expression or an array expression.

NOTE 7.13

The following is an example of a scalar expression:

```
Q + 2.3 * R
```

where Q and R are scalars.

The following is an example of an array expression:

```
A (1:10) + B (2:11)
```

where A and B are arrays.

7.1.6.1 Constant expression

A **constant expression** 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 the bounds and strides of each implied-DO are expressions whose primaries are constant expressions,

(3) A structure constructor where each component is a constant expression,

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

(5) A transformational intrinsic function reference where each argument is a constant expression,

(6) A reference to the transformational intrinsic function NULL,

(7) A reference to an intrinsic function that is

  (a) an array inquiry function (13.11.15) other than ALLOCATED,

  (b) the bit inquiry function BIT_SIZE,

  (c) the character inquiry function LEN,

  (d) the kind inquiry function KIND, or

  (e) a numeric inquiry function (13.11.8)

  and where each argument of the function is

  (a) a constant expression or

  (b) a variable whose type parameters or bounds inquired about are not

    (i) assumed,

    (ii) defined by an expression that is not a constant expression, or

    (iii) definable by an ALLOCATE or pointer assignment statement,

(8) An implied-DO variable within an array constructor where the bounds and strides of the corresponding implied-DO are constant expressions, or

(9) A constant expression enclosed in parentheses,

and where each subscript, section subscript, substring starting point, and substring ending point is a constant expression.

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

An **initialization expression** is a constant expression in which the exponentiation operation is permitted only with an integer power, and each primary is

(1) A constant or subobject of a constant,

(2) An array constructor where each element and the bounds and strides of each implied-DO are expressions whose primaries are initialization expressions,

(3) A structure constructor where each component is an initialization expression,

(4) An elemental intrinsic function reference of type integer or character where each argument is an initialization expression of type integer or character,

(5) A reference to one of the transformational functions REPEAT, RESHAPE, SELECTED_INT_KIND, SELECTED_REAL_KIND, TRANSFER, or TRIM, where each argument is an initialization expression,

(6) A reference to the transformational intrinsic function NULL,

(7) A reference to an intrinsic function that is

  (a) an array inquiry function (13.11.15) other than ALLOCATED,

  (b) the bit inquiry function BIT_SIZE,

  (c) the character inquiry function LEN,

  (d) the kind inquiry function KIND, or

  (e) a numeric inquiry function (13.11.8)

  and where each argument of the function is

  (a) an initialization expression or

  (b) a variable whose properties inquired about are not

    (i) assumed,

    (ii) defined by an expression that is not an initialization expression, or

    (iii) definable by an ALLOCATE or pointer assignment statement,

(8) An implied-DO variable within an array constructor where the bounds and strides of the corresponding implied-DO are initialization expressions, or

(9) An initialization expression enclosed in parentheses,

and where each subscript, section subscript, substring starting point, and substring ending point is an initialization expression.

R730 *initialization-expr* **is** *expr*

Constraint: *initialization-expr* shall be an initialization expression.

R731 *char-initialization-expr* **is** *char-expr*

Constraint: *char-initialization-expr* shall be an initialization expression.

R732 *int-initialization-expr* **is** *int-expr*

Constraint: *int-initialization-expr* shall be an initialization expression.

R733 *logical-initialization-expr* **is** *logical-expr*

Constraint: *logical-initialization-expr* shall be an initialization expression.

If an initialization expression includes a reference to an inquiry function for a type parameter or an array bound of an object 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 inquiry function in the same statement.

NOTE 7.14

The following are examples of constant expressions:

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

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

The following are examples of constant expressions that are not initialization expressions:

```
ABS (9.0)                                  ! Not an integer argument
3.0 ** 2.0                                 ! Not an integer power
DOT_PRODUCT ( (/ 2, 3 /), (/ 1, 7 /) )  ! Not an allowed function
```

7.1.6.2 Specification expression

A **specification expression** is an expression with limitations that make it suitable for use in specifications such as character lengths (R510) and array bounds (R515, R516). A **constant specification expression** is a specification expression that is also a constant expression.

R734 *specification-expr* **is** *scalar-int-expr*

Constraint: 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) A variable that is a dummy argument that has neither the OPTIONAL nor the INTENT (OUT) attribute, or a variable that is a subobject of such a dummy argument,

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

(4) A variable that is made accessible by use association or host association or a variable that is a subobject of such a variable,

(5) An array constructor where each element and the bounds and strides of each implied-DO are expressions whose primaries are restricted expressions,

(6) A structure constructor where each component is a restricted expression,

(7) A reference to an intrinsic function that is

  (a) an array inquiry function (13.11.15) other than ALLOCATED,

  (b) the bit inquiry function BIT_SIZE,

  (c) the character inquiry function LEN,

  (d) the kind inquiry function KIND, or

  (e) a numeric inquiry function (13.11.8)

  and where each argument of the function 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) defined by an expression that is not a restricted expression, or

(iii) definable by an ALLOCATE or pointer assignment statement,

(8) A reference to any other intrinsic function defined in this standard where each argument is a restricted expression,

(9) A reference to a specification function where each argument is a restricted expression,

(10) An implied-DO variable within an array constructor where the bounds and strides of the corresponding implied-DO are restricted expressions, or

(11) A restricted expression enclosed in parentheses,

and where any subscript, section subscript, substring starting point, or substring ending point is a restricted expression.

A function is a **specification function** if it is a pure function, is not an intrinsic function, is not an internal function, is not a statement function, does not have a dummy procedure argument, and is not defined with the RECURSIVE keyword.

NOTE 7.15

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*. Some requirement against direct recursion is necessary: since specification expressions must be evaluated before the first executable statement, there would be no way to break such a recursion. Indirect recursion in specification functions appears to be possible but difficult to implement, and of little value to the user, and so there is a general prohibition against recursive specification functions.

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, or 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 reference to an inquiry function for 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 inquiry function reference in the same statement. 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 7.16

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 Evaluation of operations

This section applies to both intrinsic and defined operations.

Any variable or function reference used as an operand in an expression shall be defined at the time the reference is executed. If the operand is a pointer, it shall be associated with a target object that is defined. All of the characters in a character data object reference shall be defined.

When a reference to an array or an array section is made, all of the selected elements shall be defined. When a structure is referenced, all of the components shall be defined.

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 (7.5.3.1), or the subscripts and strides in a FORALL statement (7.5.4) is permitted to define variables in the statement that is conditionally executed.

NOTE 7.17

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 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 the bounds and stride of any array constructor 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 7.18

For example, the array expression

```
A + B
```

produces an array the same shape as A and B. The individual array elements of the result have the values of the first element of A added to the first element of B, the second element of A added to the second element of B, etc.

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.

7.1.7.1 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 7.19

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) + X
```

where X is of size zero and W is a function, the function reference W (Z) need not be evaluated.

If a statement contains a function reference in a part of an expression that need not be evaluated, all entities that would have become defined in the execution of that reference become undefined at the completion of evaluation of the expression containing the function reference.

NOTE 7.20

In the preceding examples, evaluation of these expressions causes Z to become undefined if L or W defines its argument.

7.1.7.2 Integrity of parentheses

The sections that follow state certain conditions under which a processor may evaluate an expression that is different from the one specified by applying the rules given in 7.1.1, 7.2, and 7.3. However, any expression in parentheses shall be treated as a data entity.

NOTE 7.21

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 shall not evaluate the mathematically equivalent expression (A + B) – C.

7.1.7.3 Evaluation of numeric intrinsic operations

The rules given in 7.2.1 specify the interpretation of a numeric intrinsic operation. Once the interpretation has been established in accordance with those rules, the processor may evaluate any mathematically equivalent expression, provided that the integrity of parentheses is not violated.

Two expressions of a numeric type are mathematically equivalent if, for all possible values of their primaries, their mathematical values are equal. However, mathematically equivalent expressions of numeric type may produce different computational results.

NOTE 7.22

Any difference between the values of the expressions (1./3.)*3. and 1. is a computational difference, not a mathematical difference.

The mathematical definition of integer division is given in 7.2.1.1.

NOTE 7.23

The difference between the values of the expressions 5/2 and 5./2. is a mathematical difference, not a computational difference.

NOTE 7.24

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	Nonallowable 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 7.25

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 data type that may depend on the order of evaluation used by the processor.

NOTE 7.26

For example, in the evaluation of the expression

```
Z + R + I
```

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

7.1.7.4 Evaluation of 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 7.27

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.

7.1.7.5 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 7.28

For example, the processor may choose to evaluate the expression

```
I .GT. J
```

where I and J are integer variables, as

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

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

7.1.7.6 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 7.29

For example, for the variables L1, L2, and L3 of type logical, the processor may choose to evaluate the expression

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

as

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

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

7.1.7.7 Evaluation of a defined operation

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

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

7.2 Interpretation of intrinsic operations

The intrinsic operations are those defined in 7.1.2. These operations are divided into the following categories: numeric, 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 type, type parameters, and interpretation of an expression that consists of an intrinsic unary or binary operation are independent of the context in which the expression appears. In particular, the type, type parameters, and interpretation of such an expression are independent of the type and type parameters of any other larger expression in which it appears.

NOTE 7.30

For example, if X is of type real, J is of type integer, and INT is the real-to-integer intrinsic conversion function, the expression INT (X + J) is an integer expression and X + J is a real expression.

7.2.1 Numeric intrinsic operations

A numeric operation is used to express a numeric computation. Evaluation of a numeric operation produces a numeric value. The permitted data types 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 7.3, 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 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 depends on the data 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 7.31

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 7.32

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 7.4, 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 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 7.33

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 data types for operands of the relational intrinsic operators are specified in 7.1.2.

NOTE 7.34

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 7.5, where x_1 denotes the operand to the left of the operator and x_2 denotes the operand to the right of the operator. The operators <, <=, >, >=, ==, and /= always have the same interpretations as the operators **.**LT**.**, **.**LE**.**, **.**GT**.**, **.**GE**.**, **.**EQ**.**, and **.**NE**.**, respectively.

Table 7.5 Interpretation of the relational intrinsic operators

Operator	Representing	Use of operator	Interpretation
.LT**.**	Less than	x_1 **.**LT**.** x_2	x_1 less than x_2
<	Less than	$x_1 < x_2$	x_1 less than x_2
.LE**.**	Less than or equal to	x_1 **.**LE**.** x_2	x_1 less than or equal to x_2
<=	Less than or equal to	$x_1 <= x_2$	x_1 less than or equal to x_2
.GT**.**	Greater than	x_1 **.**GT**.** x_2	x_1 greater than x_2
>	Greater than	$x_1 > x_2$	x_1 greater than x_2
.GE**.**	Greater than or equal to	x_1 **.**GE**.** x_2	x_1 greater than or equal to x_2
>=	Greater than or equal to	$x_1 >= x_2$	x_1 greater than or equal to x_2
.EQ**.**	Equal to	x_1 .EQ**.** x_2	x_1 equal to x_2
==	Equal to	$x_1 == x_2$	x_1 equal to x_2
.NE**.**	Not equal to	x_1 **.**NE**.** x_2	x_1 not equal to x_2
/=	Not equal to	$x_1 /= x_2$	x_1 not equal to x_2

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

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 (4.3.2.1.1); 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 7.35

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 data 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 7.6, where x_1 denotes the operand to the left of the operator and x_2 denotes the operand to the right of the operator.

Table 7.6 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 non-equivalence	x_1 .NEQV. x_2	True if either x_1 or x_2 is true, but not both
.EQV.	Logical equivalence	x_1 .EQV. x_2	True if both x_1 and x_2 are true or both are false

The values of the logical intrinsic operations are shown in Table 7.7.

Table 7.7 The values of operations involving logical intrinsic operators

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

7.3 Interpretation of defined operations

The interpretation of a defined operation is provided by the function that defines the operation. The type, type parameters, and interpretation of an expression that consists of a defined operation are independent of the type and type parameters of any larger expression in which it appears. The operators <, <=, >, >=, ==, and /= always have the same interpretations as the operators .LT., .LE., .GT., .GE., .EQ., and .NE., respectively.

7.3.1 Unary defined operation

A function defines the unary operation *op* x_2 if

(1) The function is specified with a FUNCTION (12.5.2.2) or ENTRY (12.5.2.5) statement that specifies one dummy argument d_2,

(2) An interface block (12.3.2.1) provides the function with a *generic-spec* of OPERATOR (*op*),

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

(4) The type parameters, if any, of x_2 match those of d_2, and

(5) Either

  (a) The function is elemental or

  (b) The rank of x_2, and its shape if it is an array, match those of d_2.

7.3.2 Binary defined operation

A function defines the binary operation x_1 *op* x_2 if

(1) The function is specified with a FUNCTION (12.5.2.2) or ENTRY (12.5.2.5) statement that specifies two dummy arguments, d_1 and d_2,

(2) An interface block (12.3.2.1) provides the function with a *generic-spec* of OPERATOR (*op*),

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

(4) The type parameters, if any, of x_1 and x_2 match those of d_1 and d_2, respectively, and

(5) Either

(a) The function is elemental and x_1 and x_2 are conformable or

(b) The ranks of x_1 and x_2, and their shapes if either or both are arrays, match those of d_1 and d_2, respectively.

7.4 Precedence of operators

There is a precedence among the intrinsic and extension operations implied by the general form in 7.1.1, which determines the order in which the operands are combined, unless the order is changed by the use of parentheses. This precedence order is summarized in Table 7.8.

Table 7.8 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

The precedence of a defined operation is that of its operator.

NOTE 7.36

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 7.37

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 7.38

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

```
L .OR. ((A + B) >= C)
```

NOTE 7.38 *(Continued)*

For example, 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.

7.5 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.

7.5.1 Assignment statement

A variable may be defined or redefined by execution of an assignment statement.

7.5.1.1 General form

R735 *assignment-stmt* **is** *variable = expr*

where *variable* is defined in R601 and *expr* is defined in R723.

Constraint: A variable in an *assignment-stmt* shall not be a whole assumed-size array.

NOTE 7.39

Examples of an assignment statement are:

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

An assignment statement is either intrinsic or defined.

7.5.1.2 Intrinsic assignment statement

An **intrinsic assignment statement** is an assignment statement where the shapes of *variable* and *expr* conform and where

(1) The types of *variable* and *expr* are intrinsic, as specified in Table 7.9 for assignment, or

(2) The types of *variable* and *expr* are of the same derived type and no defined assignment exists for objects of this type.

A **numeric intrinsic assignment statement** is an intrinsic assignment statement for which *variable* and *expr* are of numeric type. A **character intrinsic assignment statement** is an intrinsic assignment statement for which *variable* and *expr* are of type character and have the same kind type parameter. 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 the same derived type, and there is no accessible interface block with an ASSIGNMENT (=) specifier for objects of this 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).

Table 7.9 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
character	character of the same kind type parameter as *variable*
logical	logical
derived type	same derived type as *variable*

7.5.1.3 Defined assignment statement

A **defined assignment statement** is an assignment statement that is not an intrinsic assignment statement, and is defined by a subroutine and an interface block (12.3.2.1) that specifies ASSIGNMENT (=). A **defined elemental assignment statement** is a defined assignment statement for which the subroutine is elemental (12.7).

7.5.1.4 Intrinsic assignment conformance rules

For an intrinsic assignment statement, *variable* and *expr* shall conform in shape, and if *expr* is an array, *variable* shall also be an array. The types of *variable* and *expr* shall conform with the rules of Table 7.9.

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.

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 7.10.

Table 7.10 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 13.14.	

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* shall have the same kind type parameter value, but 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*.

NOTE 7.40

For nondefault character types, the blank padding character is processor dependent.

7.5.1.5 Interpretation of intrinsic assignments

Execution of an intrinsic assignment causes, in effect, the evaluation of the expression *expr* and all expressions within *variable* (7.1.7), the possible conversion of *expr* to the type and type parameters of *variable* (Table 7.10), 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*.

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

NOTE 7.41

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).

For example, if the value of STRING prior to the assignment was 'ABCDEF', the value following the assignment is 'AABCDF'.

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

When *variable* in an intrinsic assignment is an array, the assignment is performed element-by-element on corresponding array elements of *variable* and *expr*.

NOTE 7.42

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.

The processor may perform the element-by-element assignment in any order.

NOTE 7.43

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)
```

A derived-type intrinsic assignment is performed as if each component of *expr* were assigned to the corresponding component of *variable* using pointer assignment (7.5.2) for pointer components, and intrinsic assignment for nonpointer components. The processor may perform the component-by-component assignment in any order or by any means that has the same effect.

NOTE 7.44

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 7.44 *(Continued)*

pointer assigns D % P to C % P and assigns D % S to C % S using the numeric intrinsic assignment statement, D % T to C % T using the logical intrinsic assignment statement, D % U to C % U using the character intrinsic assignment statement, and D % V to C % V using the derived-type intrinsic assignment statement.

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 parent array not specified by the array section.

7.5.1.6 Interpretation of defined assignment statements

The interpretation of a defined assignment is provided by the subroutine that defines the operation.

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

(1) The subroutine is specified with a SUBROUTINE (12.5.2.3) or ENTRY (12.5.2.5) statement that specifies two dummy arguments, d_1 and d_2,

(2) An interface block (12.3.2.1) provides the subroutine with a *generic-spec* of ASSIGNMENT (=),

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

(4) The type parameters, if any, of x_1 and x_2 match those of d_1 and d_2, respectively, and

(5) Either

  (a) The subroutine is elemental and either x_1 and x_2 have the same shape or x_2 is scalar or

  (b) The ranks of x_1 and x_2, and their shapes if either or both are arrays, match those of d_1 and d_2, respectively.

The types of x_1 and x_2 shall not both be numeric, both be logical, or both be character with the same kind type parameter value.

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 7.45

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 same effect as if the evaluation of all operations in x_2 and x_1 occurs before any portion of x_1 is defined.

7.5.2 Pointer assignment

Pointer assignment causes a pointer to become associated with a target or causes its pointer association status to become disassociated or undefined.

R736 *pointer-assignment-stmt* **is** *pointer-object => target*

R737 *target* **is** *variable*
or *expr*

Constraint: The *pointer-object* shall have the POINTER attribute.

Constraint: The *variable* shall have the TARGET attribute or be a subobject of an object with the TARGET attribute, or it shall have the POINTER attribute.

Constraint: The *target* shall be of the same type, kind type parameters, and rank as the pointer.

Constraint: The *target* shall not be an array section with a vector subscript.

Constraint: The *expr* shall deliver a pointer result.

The *target* shall have the same type parameters as the *pointer-object*. If the *target* is not a pointer, the pointer assignment statement associates the *pointer-object* with the *target*. If the *target* is a pointer that is associated, the *pointer-object* is associated with the same object as the *target*. If the *target* is a pointer that is disassociated or a reference to the NULL intrinsic function, the *pointer-object* becomes disassociated. If the *target* is a pointer with undefined association status, the *pointer-object* acquires an undefined association status.

Any previous association between the *pointer-object* and a target is broken.

Pointer assignment for a pointer component of a structure also may take place by execution of a derived-type intrinsic assignment statement (7.5.1.5) or a defined assignment statement (7.5.1.6).

In addition to pointer assignment, a pointer becomes associated with a target by allocation of the pointer.

A pointer shall not be referenced or defined unless it is associated with a target that may be referenced or defined.

NOTE 7.46

The following are examples of pointer assignment statements.

```
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)
```

7.5.3 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.

7.5.3.1 General form of the masked array assignment

A **masked array assignment** is either a WHERE statement or WHERE construct.

R738 *where-stmt* **is** WHERE (*mask-expr*) *where-assignment-stmt*

R739 *where-construct* **is** *where-construct-stmt*
[*where-body-construct*] ...
[*masked-elsewhere-stmt*
[*where-body-construct*] ...] ...
[*elsewhere-stmt*
[*where-body-construct*] ...]
end-where-stmt

R740 *where-construct-stmt* **is** [*where-construct-name*:] WHERE (*mask-expr*)

R741 *where-body-construct* **is** *where-assignment-stmt*
or *where-stmt*
or *where-construct*

R742 *where-assignment-stmt* **is** *assignment-stmt*

R743 *mask-expr* **is** *logical-expr*

R744 *masked-elsewhere-stmt* **is** ELSEWHERE (*mask-expr*) [*where-construct-name*]

R745 *elsewhere-stmt* **is** ELSEWHERE [*where-construct-name*]

R746 *end-where-stmt* **is** END WHERE [*where-construct-name*]

Constraint: A *where-assignment-stmt* that is a defined assignment shall be elemental.

Constraint: 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*.

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 7.47

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
```

7.5.3.2 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 only 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 7.48

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 statement or 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 only once.

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*.

A statement that is part of a *where-body-construct* shall not be a branch target statement. 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. Execution of an END WHERE has no effect.

NOTE 7.49

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
```

7.5.4 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.

7.5.4.1 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*.

R747 *forall-construct* **is** *forall-construct-stmt*
[*forall-body-construct*] ...
end-forall-stmt

R748 *forall-construct-stmt* **is** [*forall-construct-name* :] FORALL *forall-header*

R749 *forall-header* **is** (*forall-triplet-spec-list* [, *scalar-mask-expr*])

R750 *forall-triplet-spec* **is** *index-name* = *subscript* : *subscript* [: *stride*]

R617 *subscript* **is** *scalar-int-expr*

R603 *stride* **is** *scalar-int-expr*

R751 *forall-body-construct* **is** *forall-assignment-stmt*
or *where-stmt*
or *where-construct*
or *forall-construct*
or *forall-stmt*

R752 *forall-assignment-stmt* **is** *assignment-stmt*
or *pointer-assignment-stmt*

R753 *end-forall-stmt* **is** END FORALL [*forall-construct-name*]

Constraint: 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*.

Constraint: The *scalar-mask-expr* shall be scalar and of type logical.

Constraint: Any procedure referenced in the *scalar-mask-expr,* including one referenced by a defined operation, shall be a pure procedure (12.6).

Constraint: The *index-name* shall be a named scalar variable of type integer.

Constraint: 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.

Constraint: A statement in a *forall-body-construct* shall not define an *index-name* of the *forall-construct*.

Constraint: Any procedure referenced in a *forall-body-construct*, including one referenced by a defined operation or assignment, shall be a pure procedure.

Constraint: A *forall-body-construct* shall not be a branch target.

NOTE 7.50

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 7.51

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 7.52

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)
   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 (14.1.3). 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 7.53

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
```

7.5.4.2 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.

7.5.4.2.1 Determination of the values for *index-name* 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 7.10). 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 $max \le 0$ for some *index-name*, the execution of the construct is complete. Otherwise, the set of values for the *index-name* is

$$m_1 + (k-1) \times m_3 \qquad \text{where } k = 1, 2, \ldots, max.$$

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 7.54

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

7.5.4.2.2 Evaluation of the *scalar-mask-expr*

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* value**s is defined to be the subset of all possible combinations (7.5.4.2.1) for which the *scalar-mask-expr* has the value .TRUE..

NOTE 7.55

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)
. . .
```

7.5.4.2.3 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 *target* and *pointer-object*, the determination of any pointers within *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 *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 or *pointer-object* that becomes associated by the statement.

NOTE 7.56

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.

```
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) .NE. 0.0)
   X(I) = 1.0 / Y(I)
END FORALL
```

Each statement in a *where-construct* (7.5.3) 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 7.57

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,:) .EQ. 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 7.58

This FORALL construct contains a nested FORALL construct. It assigns the transpose of the lower triangle of array A (the section below the main diagonal) to the 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
```

7.5.4.3 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.

R754 *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 (14.1.3).

NOTE 7.59

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.

```
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 7.58 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.

7.5.4.4 Restrictions on FORALL constructs and statements

A many-to-one assignment is more than one assignment to the same object or subobject, 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 7.60

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 and may be used to control the execution sequence:

(1) IF Construct
(2) CASE Construct
(3) DO 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.

NOTE 8.3

For example, if a statement inside the block has a statement label, a GO TO statement using that label is only allowed to appear in the same block.

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. Unless there is a transfer of control out of the block, the execution of the block is completed when the last executable construct in the sequence is executed. The action that takes place at the terminal boundary depends on the 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 no more than one of its constituent blocks. The **IF statement** controls the execution of a single statement (8.1.2.4).

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*]

Constraint: If the *if-then-stmt* of an *if-construct* is identified by 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* is not identified by an *if-construct-name*, the corresponding *end-if-stmt* shall not specify an *if-construct-name*. If an *else-if-stmt* or *else-stmt* is identified by 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 will be 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 the IF construct. Execution of an END IF statement has no effect.

8.1.2.3 Examples of IF constructs

NOTE 8.4

```
IF (CVAR .EQ. '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 .GT. 0) THEN
   B = C/A
   IF (B .GT. 0) THEN
      D = 1.0
   END IF
ELSE IF (C .GT. 0) THEN
   B = A/C
   D = -1.0
ELSE
   B = ABS (MAX (A, C))
   D = 0
END IF
```

8.1.2.4 IF statement

The IF statement controls a single action statement (R216).

R807 *if-stmt* **is** IF (*scalar-logical-expr*) *action-stmt*

Constraint: 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 as though a CONTINUE statement (8.3) were executed.

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.

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*]

Constraint: If the *select-case-stmt* of a *case-construct* is identified by 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* is not identified by a *case-construct-name*, the corresponding *end-select-stmt* shall not specify a *case-construct-name*. If a *case-stmt* is identified by 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

Constraint: 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*

Constraint: For a given *case-construct*, each *case-value* shall be of the same type as *case-expr*. For character type, length differences are allowed, but the kind type parameters shall be the same.

Constraint: A *case-value-range* using a colon shall not be used if *case-expr* is of type logical.

Constraint: 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 data type logical if the expression *c* **.**EQV**.** *v* is true, and a match occurs for data type integer or character if the expression *c* **.**EQ**.** *v* is true.

(2) If the case value range is of the form *low* : *high*, a match occurs if the expression *low* **.**LE**.** *c* **.**AND**.** *c* **.**LE**.** *high* is true.

(3) If the case value range is of the form *low* :, a match occurs if the expression *low* **.**LE**.** *c* is true.

(4) If the case value range is of the form : *high*, a match occurs if the expression *c* **.**LE**.** *high* is true.

(5) If no other selector matches and a DEFAULT selector is present, it matches the case index.

(6) If no other selector matches and the DEFAULT selector is absent, 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 statement only from within the 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
DO I = 1, 80
   CHECK_PARENS:  SELECT CASE (LINE (I:I))
   CASE ('(')
      LEVEL = LEVEL + 1
   CASE (')')
      LEVEL = LEVEL - 1
      IF (LEVEL .LT. 0) THEN
         PRINT *, 'UNEXPECTED RIGHT PARENTHESIS'
         EXIT
      END IF
   CASE DEFAULT
      ! Ignore all other characters
   END SELECT CHECK_PARENS
END DO
IF (LEVEL .GT. 0) THEN
   PRINT *, 'MISSING RIGHT PARENTHESIS'
END IF
```

NOTE 8.8

The following three fragments are equivalent:

```
IF (SILLY .EQ. 1) THEN
   CALL THIS
ELSE
   CALL THAT
END IF
```

```
SELECT CASE (SILLY .EQ. 1)
CASE (.TRUE.)
   CALL THIS
CASE (.FALSE.)
   CALL THAT
END SELECT
```

NOTE 8.8 *(Continued)*

```
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 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 (8.1.4.4.4) anywhere in the DO construct may be executed to terminate the loop immediately. A particular iteration of the loop may be curtailed by executing a CYCLE statement (8.1.4.4.3).

8.1.4.1 Forms of the DO construct

The DO construct may be written in either a block form or a nonblock form.

R816 *do-construct* **is** *block-do-construct*
or *nonblock-do-construct*

8.1.4.1.1 Form of the block DO construct

R817 *block-do-construct* **is** *do-stmt*
do-block
end-do

R818 *do-stmt* **is** *label-do-stmt*
or *nonlabel-do-stmt*

R819 *label-do-stmt* **is** [*do-construct-name* :] DO *label* [*loop-control*]

R820 *nonlabel-do-stmt* **is** [*do-construct-name* :] DO [*loop-control*]

R821 *loop-control* **is** [,] *do-variable* = *scalar-int-expr*, *scalar-int-expr* ■
■ [, *scalar-int-expr*]
or [,] WHILE (*scalar-logical-expr*)

R822 *do-variable* **is** *scalar-int-variable*

Constraint: The *do-variable* shall be a named scalar variable of type integer.

R823 *do-block* **is** *block*

R824 *end-do* **is** *end-do-stmt*
or *continue-stmt*

R825 *end-do-stmt* **is** END DO [*do-construct-name*]

Constraint: If the *do-stmt* of a *block-do-construct* is identified by 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* is not identified by a *do-construct-name*, the corresponding *end-do* shall not specify a *do-construct-name*.

Constraint: If the *do-stmt* is a *nonlabel-do-stmt*, the corresponding *end-do* shall be an *end-do-stmt*.

Constraint: If the *do-stmt* is a *label-do-stmt*, the corresponding *end-do* shall be identified with the same *label*.

8.1.4.1.2 Form of the nonblock DO construct

R826 *nonblock-do-construct* **is** *action-term-do-construct*
or *outer-shared-do-construct*

R827 *action-term-do-construct* **is** *label-do-stmt*
do-body
do-term-action-stmt

R828 *do-body* **is** [*execution-part-construct*] ...

R829 *do-term-action-stmt* **is** *action-stmt*

Constraint: 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*.

Constraint: The *do-term-action-stmt* shall be identified with a label and the corresponding *label-do-stmt* shall refer to the same label.

R830 *outer-shared-do-construct* **is** *label-do-stmt*
do-body
shared-term-do-construct

R831 *shared-term-do-construct* **is** *outer-shared-do-construct*
or *inner-shared-do-construct*

R832 *inner-shared-do-construct* **is** *label-do-stmt*
do-body
do-term-shared-stmt

R833 *do-term-shared-stmt* **is** *action-stmt*

Constraint: 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*.

Constraint: The *do-term-shared-stmt* shall be identified with a label and all of the *label-do-stmt*s of the *shared-term-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.

8.1.4.2 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*.

8.1.4.3 Active and inactive DO constructs

A DO construct is either **active** or **inactive**. Initially inactive, a DO construct becomes active only when its DO statement is executed.

Once active, the DO construct becomes inactive only when the construct it specifies is terminated (8.1.4.4.4). When an active DO construct becomes inactive, the *do-variable*, if any, retains its last defined value.

8.1.4.4 Execution of a DO construct

A DO construct specifies a loop, 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.

8.1.4.4.1 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 7.10). If *scalar-int-expr*$_3$ does not appear, m_3 has the value 1. The value 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 8.10

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.

8.1.4.4.2 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.

8.1.4.4.3 CYCLE statement

Step (2) in the above execution cycle may be curtailed by executing a CYCLE statement from within the range of the loop.

R834 *cycle-stmt* **is** CYCLE [*do-construct-name*]

Constraint: 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.

8.1.4.4.4 Loop termination

The EXIT statement provides one way of terminating a loop.

R835 *exit-stmt* **is** EXIT [*do-construct-name*]

Constraint: 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.

8.1.4.5 Examples of DO constructs

NOTE 8.11

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 8.12

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 .EQ. 0)
   IF (X .GE. 0.) THEN
      CALL SUBA (X)
      CALL SUBB (X)
           ...
      CALL SUBZ (X)
   ENDIF
   READ (IUN, '(1X, G14.7)', IOSTAT = IOS) X
END DO
```

NOTE 8.13

The following example behaves exactly the same as the one in Note 8.12. 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 .NE. 0) EXIT
   IF (X < 0.) CYCLE
   CALL SUBA (X)
   CALL SUBB (X)
      . . .
   CALL SUBZ (X)
END DO
```

NOTE 8.14

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. A **branch target statement** is an *action-stmt*, an *if-then-stmt*, an *end-if-stmt*, a *select-case-stmt*, an *end-select-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*.

It is permissible to branch to an END SELECT statement only from within its CASE construct.

It is permissible to branch to an END IF statement only from within its IF construct.

It is permissible to branch to an *end-do-stmt* or a *do-term-action-stmt* only from within its DO construct. It is permissible to branch to a *do-term-shared-stmt* only from within its *inner-shared-do-construct*.

8.2.1 Statement labels

A **statement label** provides a means of referring to an individual statement. Only branch target statements (8.2), FORMAT statements, and DO terminations shall be referred to by the use of statement labels (3.2.4).

8.2.2 GO TO statement

R836 *goto-stmt* **is** GO TO *label*

Constraint: 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.

8.2.3 Computed GO TO statement

R837 *computed-goto-stmt* **is** GO TO (*label-list*) [,] *scalar-int-expr*

Constraint: 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*.

The same statement label may appear more than once in a label list.

Execution of a computed GO TO statement causes evaluation of the scalar integer expression. If this value is i such that $1 \le i \le n$ where n is the number of labels in *label-list*, a transfer of control occurs so that the next statement executed is the one identified by the 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.

8.2.4 Arithmetic IF statement

R838 *arithmetic-if-stmt* **is** IF (*scalar-numeric-expr*) *label , label , label*

Constraint: Each *label* shall be the label of a branch target statement that appears in the same scoping unit as the *arithmetic-if-stmt*.

Constraint: The *scalar-numeric-expr* shall not be of type complex.

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 as the value of the numeric expression is less than zero, equal to zero, or greater than zero, respectively.

8.3 CONTINUE statement

Execution of a CONTINUE statement has no effect.

R839 *continue-stmt* **is** CONTINUE

8.4 STOP statement

R840 *stop-stmt* **is** STOP [*stop-code*]

R841 *stop-code* **is** *scalar-char-constant*
or *digit* [*digit* [*digit* [*digit* [*digit*]]]]

Constraint: *scalar-char-constant* shall be of type default character.

Execution of a STOP statement causes termination 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.

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, and INQUIRE statements.

The READ statement is a **data transfer input statement**. The WRITE statement and the PRINT statement are **data transfer output statements**. The OPEN statement and the CLOSE statement are **file connection statements**. The INQUIRE statement is a **file inquiry statement**. The BACKSPACE, ENDFILE, and REWIND statements are **file positioning statements**.

9.1 Records

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; for example, by some manual input device.

9.1.2 Unformatted record

An **unformatted record** consists of a sequence of values in a processor-dependent form and may contain data of any type or may contain no data. The length of an unformatted record is measured in processor-dependent units and depends on the output list (9.4.2) used when it is written, as well as on the processor and the external medium. The length may be zero. Unformatted records may be read or written only by unformatted input/output statements.

9.1.3 Endfile record

An **endfile record** is written explicitly by the ENDFILE statement; the file 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 (file) 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 (9.5.1).

9.2 Files

A **file** is a sequence of records.

There are two kinds of files:

(1) External

(2) Internal

NOTE 9.3

For more explanatory information on files, see C.6.1.

9.2.1 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 9.4

An example of a restriction on input/output statements (9.8) is that an input statement shall not specify that data are to be read from a printer.

A file may have a name; a file that has a name is called a **named file**. The name of a named file is a character string. The set of allowable names for a file is processor dependent.

An external file that is connected to a unit has a **position** property (9.2.1.3).

9.2.1.1 File existence

At any given time, there is a processor-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, 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.

9.2.1.2 File access

There are two methods of accessing the records of an external file, sequential and direct. Some files may have more than one allowed access method; other files may be restricted to one access method.

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 the file is determined when the file is connected to a unit (9.3.2) or when the file is created if the file is preconnected (9.3.3).

9.2.1.2.1 Sequential access

When connected for **sequential access**, an external file has the following properties:

(1) The order of the records is the order in which they were written if the direct access method is not a member of the set of allowed access methods for the file. If the direct access method is also a member of the set of allowed access methods for the file, the order of the records is the same as that specified for direct access. In this case, the first record 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 or a file positioning statement, the last record, if any, of the file shall be an endfile record.

(3) The records of the file shall not be read or written by direct access input/output statements.

9.2.1.2.2 Direct access

When connected for **direct access**, an external file has the following properties:

(1) Each record of the file is uniquely identified by a positive integer called the **record number**. The record number of a record is specified when the record is written. Once established, the record number of a record can never be changed. The order of the records is the order of their record numbers.

NOTE 9.7

A record may not 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) Reading and writing records is accomplished only by direct access input/output statements.

(4) All records of the file have the same length.

(5) Records need not be read or written in the order of their record numbers. Any record may be written into the file while it is connected to a unit. For example, it is permissible to write record 3, even though records 1 and 2 have not been written. Any record may be read from the file while it is connected to a unit, provided that the record has been written since the file was created.

(6) The records of the file shall not be read or written using list-directed formatting (10.8), namelist formatting (10.9), or a nonadvancing input/output statement (9.2.1.3.1).

9.2.1.3 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. The **terminal point** is the position just after the last record. If there are no records in the file, the initial point and the terminal point are the same position.

If a file is positioned within a record, that record is the **current record**; otherwise, there is no current record.

Let n be the number of records in the file. If $1 < i \leq n$ and a file is positioned within the ith record or between the $(i-1)$th record and the ith record, the $(i-1)$th record is the **preceding record**. If $n \geq 1$ and the file is positioned at its terminal point, the preceding record is the nth and last record. If $n = 0$ or if a file is positioned at its initial point or within the first record, there is no preceding record.

If $1 \leq i < n$ and a file is positioned within the ith record or between the ith and $(i+1)$th record, the $(i+1)$th record is the **next record**. If $n \geq 1$ and the file is positioned at its initial point, the first record is the next record. If $n = 0$ or if a file is positioned at its terminal point or within the nth (last) record, there is no next record.

9.2.1.3.1 Advancing and nonadvancing input/output

An **advancing input/output statement** always positions the file after the last record read or written, unless there is an error condition.

A **nonadvancing input/output statement** may position the file at a character position within the current record, or a subsequent record (10.6.2). 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.

9.2.1.3.2 File position prior to data transfer

The positioning of the file prior to data transfer depends on the method of access: sequential or direct.

For sequential access on input, 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 record specifier. This record becomes the current record.

9.2.1.3.3 File position after data transfer

If an error condition (9.4.3) occurred, the position of the file is indeterminate. If no error condition occurred, but an end-of-file condition (9.4.3) occurred as a result of reading an endfile record, the file is positioned after the endfile record.

For nonadvancing input, if no error condition or end-of-file condition occurred, but an end-of-record condition (9.4.3) 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.

9.2.2 Internal files

Internal files provide a means of transferring and converting data from internal storage to internal storage.

9.2.2.1 Internal file properties

An internal file has the following properties:

(1) The file is a variable of default 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. This record becomes the current record.

(8) On input, blanks are treated in the same way as for an external file opened with a BLANK= specifier having the value NULL and records are padded with blanks if necessary (9.4.4.4.2).

(9) On list-directed output, character constants are not delimited (10.8.2).

9.2.2.2 Internal file restrictions

An internal file has the following restrictions:

(1) Reading and writing records shall be accomplished only by sequential access formatted input/output statements that do not specify namelist formatting.

(2) An internal file shall not be specified in a file connection statement, a file positioning statement, or a file inquiry statement.

9.3 File connection

A **unit**, specified by an *io-unit*, provides a means for referring to a file.

R901 *io-unit* **is** *external-file-unit*
or *
or *internal-file-unit*

R902 *external-file-unit* **is** *scalar-int-expr*

R903 *internal-file-unit* **is** *default-char-variable*

Constraint: The *default-char-variable* shall not be an array section with a vector subscript.

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 *external-file-unit* or an asterisk. An **internal unit** is used to refer to an internal file and is specified by an *internal-file-unit*.

If a character variable that identifies an internal file unit is a pointer, it shall be associated. If the character variable is an allocatable array or a subobject of such an array, the array shall be currently allocated.

A scalar integer expression that identifies an external file unit shall be zero or positive.

The *io-unit* in a file positioning statement, a file connection statement, or a file inquiry statement shall be an *external-file-unit*.

The external unit identified by the value of the *scalar-int-expr* is the same external unit in all program units of the program.

NOTE 9.8

In the example:

```
SUBROUTINE A
   READ (6) X
      ...
SUBROUTINE B
   N = 6
   REWIND N
```

the value 6 used in both program units identifies the same external unit.

An asterisk identifies particular processor-dependent external units that are preconnected for formatted sequential access (9.4.4.2).

9.3.1 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 INQUIRE statement and the CLOSE statement also may refer to units that do not exist.

9.3.2 Connection of a file to a unit

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; if a unit is connected to a file, the file is connected to the unit.

All input/output statements except an OPEN, a CLOSE, or an INQUIRE statement 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 (9.2.1.1).

NOTE 9.9

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.

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 9.10

The only means of referencing a file that has been disconnected is by the appearance of its name in an OPEN or INQUIRE statement. There may be no means of reconnecting an unnamed file once it is disconnected.

An internal unit is always connected to the internal file designated by the variable of default character type that identifies the unit.

NOTE 9.11

For more explanatory information on file connection properties, see C.6.3.

9.3.3 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.

9.3.4 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 specifiers of a connection between a file and a unit.

An external unit may be connected by an OPEN statement in any program unit of 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 properties specified by an OPEN statement become a part of the connection.

If the file to be connected to the unit is not the same as the file to which the unit is connected, the effect is as if a CLOSE statement without a STATUS= specifier had been executed for the unit immediately prior to the execution of an OPEN statement.

If the file to be connected to the unit is the same as the file to which the unit is connected, only the BLANK=, DELIM=, PAD=, ERR=, and IOSTAT= specifiers may have values different from those

currently in effect. If the POSITION= specifier is present 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 a value of OLD. Execution of such an OPEN statement causes any new value of the BLANK=, DELIM=, or PAD= specifiers 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= and IOSTAT= 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 =] *external-file-unit*
or IOSTAT = *scalar-default-int-variable*
or ERR = *label*
or FILE = *file-name-expr*
or STATUS = *scalar-default-char-expr*
or ACCESS = *scalar-default-char-expr*
or FORM = *scalar-default-char-expr*
or RECL = *scalar-int-expr*
or BLANK = *scalar-default-char-expr*
or POSITION = *scalar-default-char-expr*
or ACTION = *scalar-default-char-expr*
or DELIM = *scalar-default-char-expr*
or PAD = *scalar-default-char-expr*

R906 *file-name-expr* **is** *scalar-default-char-expr*

Constraint: If the optional characters UNIT= are omitted from the unit specifier, the unit specifier shall be the first item in the *connect-spec-list*.

Constraint: Each specifier shall not appear more than once in a given *open-stmt*; an *external-file-unit* shall be specified.

Constraint: 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 be present. If the STATUS= specifier has the value SCRATCH, the FILE= specifier shall be absent. If the STATUS= specifier has the value OLD, the FILE= specifier shall be present unless the unit is currently 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. If a processor is capable of representing letters in both upper and lower case, the value specified is without regard to case. Some specifiers have a default value if the specifier is omitted.

The IOSTAT= specifier and ERR= specifier are described in 9.4.1.4 and 9.4.1.5, respectively.

NOTE 9.12

An example of an OPEN statement is:

```
OPEN (10, FILE = 'employee.names',  ACTION = 'READ', PAD = 'YES')
```

NOTE 9.13

For more explanatory information on the OPEN statement, see C.6.2.

9.3.4.1 FILE= specifier in the OPEN statement

The value of the FILE= specifier is the name of the file to be connected to the specified unit. 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. If a processor is capable of representing letters in both upper and lower case, the interpretation of case is processor dependent.

9.3.4.2 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 termination of the program.

NOTE 9.14

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.

9.3.4.3 ACCESS= specifier in the OPEN statement

The *scalar-default-char-expr* shall evaluate to SEQUENTIAL or DIRECT. The ACCESS= specifier specifies the access method for the connection of the file as being sequential or direct. If this specifier is omitted, the 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.

9.3.4.4 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, 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.

9.3.4.5 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 be present 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 processor-dependent 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.

9.3.4.6 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 file being connected for formatted input/output. If NULL is specified, all blank characters in numeric formatted input fields on the specified unit are ignored, except that a field of all blanks has a value of zero. If ZERO is specified, all blanks other than leading blanks are treated as zeros. If this specifier is omitted, the default value is NULL.

9.3.4.7 POSITION= specifier in the OPEN statement

The *scalar-default-char-expr* shall evaluate to ASIS, REWIND, or APPEND. The connection shall be for sequential access. A file that did not exist previously (a new file, either specified explicitly or by default) 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.

9.3.4.8 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 I/O 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.

9.3.4.9 DELIM= specifier in the OPEN statement

The *scalar-default-char-expr* shall evaluate to APOSTROPHE, QUOTE, or NONE. If APOSTROPHE is specified, the apostrophe shall be used to delimit character values written with list-directed or namelist formatting and all internal apostrophes shall be doubled. If QUOTE is specified, the quotation mark shall be used to delimit character values written with list-directed or namelist formatting and all internal quotation marks shall be doubled. If the value of this specifier is NONE, a character value when written shall not be delimited by apostrophes or quotation marks, nor shall any internal apostrophes or quotation marks be doubled. If this specifier is omitted, the default value is NONE. This specifier is permitted only for a file being connected for formatted input/output. This specifier is ignored during input of a formatted record.

9.3.4.10 PAD= specifier in the OPEN statement

The *scalar-default-char-expr* shall evaluate to YES or NO. If YES is specified, a formatted input record is padded with blanks (9.4.4.4.2) when an input list is specified and the format specification requires more data from a record than the record contains. If NO is specified, the input list and the format specification shall not require more characters from a record than the record contains. If this specifier is omitted, the default value is YES. This specifier is permitted only for a file being connected for formatted input/output. This specifier is ignored during output of a formatted record.

NOTE 9.15

For nondefault character types, the blank padding character is processor dependent.

9.3.5 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 that refers to 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 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 termination of execution of a program for reasons other than an error condition, all units that are connected are closed. Each unit is closed with status KEEP unless the file status prior to termination of execution was SCRATCH, in which case the unit is closed with status DELETE.

NOTE 9.16

The effect is as though a CLOSE statement without a STATUS= specifier were executed on each connected unit.

R907 *close-stmt* **is** CLOSE (*close-spec-list*)

R908 *close-spec* **is** [UNIT =] *external-file-unit*
or IOSTAT = *scalar-default-int-variable*
or ERR = *label*
or STATUS = *scalar-default-char-expr*

Constraint: If the optional characters UNIT= are omitted from the unit specifier, the unit specifier shall be the first item in the *close-spec-list*.

Constraint: Each specifier shall not appear more than once in a given *close-stmt*; an *external-file-unit* shall be specified.

Constraint: 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. If a processor is capable of representing letters in both upper and lower case, the value specified is without regard to case.

The IOSTAT= specifier and ERR= specifier are described in 9.4.1.4 and 9.4.1.5, respectively.

NOTE 9.17

An example of a CLOSE statement is:

```
CLOSE (10, STATUS = 'KEEP')
```

NOTE 9.18

For more explanatory information on the CLOSE statement, see C.6.4.

9.3.5.1 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.

9.4 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.

R909 *read-stmt* **is** READ (*io-control-spec-list*) [*input-item-list*]
or READ *format* [, *input-item-list*]

R910 *write-stmt* **is** WRITE (*io-control-spec-list*) [*output-item-list*]

R911 *print-stmt* **is** PRINT *format* [, *output-item-list*]

NOTE 9.19

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)
```

9.4.1 Control information list

The *io-control-spec-list* is a **control information list** that includes

(1) A reference to the source or destination of the data to be transferred,
(2) Optional specification of editing processes,
(3) Optional specification to identify a record,
(4) Optional specification of exception handling,
(5) Optional return of status,
(6) Optional record advancing specification, and
(7) Optional return of number of characters read.

The control information list governs the data transfer.

R912 *io-control-spec* **is** [UNIT =] *io-unit*
or [FMT =] *format*
or [NML =] *namelist-group-name*
or REC = *scalar-int-expr*
or IOSTAT = *scalar-default-int-variable*
or ERR = *label*
or END = *label*
or ADVANCE = *scalar-default-char-expr*
or SIZE = *scalar-default-int-variable*
or EOR = *label*

Constraint: An *io-control-spec-list* shall contain exactly one *io-unit* and may contain at most one of each of the other specifiers.

Constraint: An END=, EOR=, or SIZE= specifier shall not appear in a *write-stmt*.

Constraint: 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.

Constraint: A *namelist-group-name* shall not be present if an *input-item-list* or an *output-item-list* is present in the data transfer statement.

Constraint: An *io-control-spec-list* shall not contain both a *format* and a *namelist-group-name*.

Constraint: If the optional characters UNIT= are omitted from the unit specifier, the unit specifier shall be the first item in the control information list.

Constraint: If the optional characters FMT= are omitted from the format specifier, the format specifier shall be the second item in the control information list and the first item shall be the unit specifier without the optional characters UNIT=.

Constraint: If the optional characters NML= are omitted from the namelist specifier, the namelist specifier shall be the second item in the control information list and the first item shall be the unit specifier without the optional characters UNIT=.

Constraint: If the unit specifier specifies an internal file, the *io-control-spec-list* shall not contain a REC= specifier or a *namelist-group-name*.

Constraint: If the REC= specifier is present, an END= specifier shall not appear, a *namelist-group-name* shall not appear, and the *format*, if any, shall not be an asterisk specifying list-directed input/output.

Constraint: An ADVANCE= specifier may be present only in a formatted sequential input/output statement with explicit format specification (10.1) whose control information list does not contain an internal file unit specifier.

Constraint: If an EOR= specifier is present, an ADVANCE= specifier also shall appear.

Constraint: If a SIZE= specifier is present, an ADVANCE= specifier also shall appear.

A SIZE= specifier may be present only in an input statement that contains an ADVANCE= specifier with the value NO.

An EOR= specifier may be present 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**.

In a data transfer statement, the variable specified in an IOSTAT= or a SIZE= specifier, if any, shall not be associated with any entity in the data transfer input/output list (9.4.2) 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= or a 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*.

For the ADVANCE= specifier, the *scalar-default-char-expr* has a limited list of character values. Any trailing blanks are ignored. If a processor is capable of representing letters in both upper and lower case, the value specified is without regard to case.

NOTE 9.20

An example of a READ statement is:

```
READ (IOSTAT = IOS, UNIT = 6, FMT = '(10F8.2)') A, B
```

9.4.1.1 Format specifier

The FMT= specifier supplies a format specification or specifies list-directed formatting for a formatted input/output statement.

R913 *format* **is** *default-char-expr*
or *label*
or *

Constraint: The *label* shall be the label of a FORMAT statement that appears in the same scoping unit as the statement containing the format specifier.

The *default-char-expr* shall evaluate to a valid format specification (10.1.1 and 10.1.2).

NOTE 9.21

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 9.22

An example in which the format is a character expression is:

```
READ (6, FMT = "(" // CHAR_FMT // ")" )  X, Y, Z
```

where CHAR_FMT is a default character variable.

9.4.1.2 Namelist specifier

The NML= specifier supplies the *namelist-group-name* (5.4). This name identifies a specific collection of data objects on which transfer is to be performed.

If a *namelist-group-name* is present, the statement is a **namelist input/output statement**.

9.4.1.3 Record number

The REC= specifier specifies the number of the record that is to be read or written. This specifier may be present only in an input/output statement that specifies a unit connected for direct access. If the control information list contains a REC= specifier, the statement is a **direct access input/output statement** ; otherwise, it is a **sequential access input/output statement**.

9.4.1.4 Input/output status

Execution of an input/output statement containing the IOSTAT= specifier causes the variable specified in the IOSTAT= specifier to become defined

(1) With a zero value if neither an error condition, an end-of-file condition, nor an end-of-record condition occurs,

(2) With a processor-dependent positive integer value if an error condition occurs,

(3) With a processor-dependent negative integer value if an end-of-file condition occurs and no error condition occurs, or

(4) With a processor-dependent negative integer value different from the end-of-file value if an end-of-record condition occurs and no error condition or end-of-file condition occurs.

NOTE 9.23

An end-of-file condition may occur only during execution of a sequential input statement and an end-of-record condition may occur only during execution of a nonadvancing input statement.

Consider the example:

```
READ (FMT = "(E8.3)", UNIT = 3, IOSTAT = IOSS) X
IF (IOSS < 0) THEN
   ! Perform end-of-file processing on the file connected to unit 3.
   CALL END_PROCESSING
ELSE IF (IOSS > 0) THEN
   ! Perform error processing
   CALL ERROR_PROCESSING
END IF
```

9.4.1.5 Error branch

If an error condition (9.4.3) occurs during execution of an input/output statement that contains an ERR= specifier

(1) Execution of the input/output statement terminates,

(2) The position of the file specified in the input/output statement becomes indeterminate,

(3) If the input/output statement also contains an IOSTAT= specifier, the variable specified becomes defined with a processor-dependent positive integer value,

(4) If the statement is a READ statement and it contains a SIZE= specifier, the variable becomes defined with an integer value (9.4.1.9), and

(5) Execution continues with the statement specified in the ERR= specifier.

9.4.1.6 End-of-file branch

If an end-of-file condition (9.4.3) occurs and no error condition (9.4.3) occurs during execution of an input statement that contains an END= specifier

(1) Execution of the input statement terminates,

(2) If the file specified in the input statement is an external file, it is positioned after the endfile record,

(3) If the input statement also contains an IOSTAT= specifier, the variable specified becomes defined with a processor-dependent negative integer value, and

(4) Execution continues with the statement specified in the END= specifier.

9.4.1.7 End-of-record branch

If an end-of-record condition (9.4.3) occurs and no error condition (9.4.3) occurs during execution of an input/output statement that contains an EOR= specifier

(1) If the PAD= specifier has the value YES, the record is padded with blanks to satisfy the input list item (9.4.4.4.2) and corresponding data edit descriptor that requires more characters than the record contains,

(2) Execution of the input statement terminates,

(3) The file specified in the input statement is positioned after the current record,

(4) If the input statement also contains an IOSTAT= specifier, the variable specified becomes defined with a processor-dependent negative integer value,

(5) If the input statement contains a SIZE= specifier, the variable becomes defined with an integer value (9.4.1.9), and

(6) Execution continues with the statement specified in the EOR= specifier.

9.4.1.8 Advance specifier

The *scalar-default-char-expr* shall evaluate to YES or NO. The ADVANCE= specifier determines whether nonadvancing input/output occurs for this input/output statement. If NO is specified, nonadvancing input/output occurs. If YES is specified, advancing formatted sequential input/output occurs. If this specifier is omitted, the default value is YES.

9.4.1.9 Character count

When a 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 (9.4.4.4.2) are not counted.

9.4.2 Data transfer input/output list

An input/output list specifies the entities whose values are transferred by a data transfer input/output statement.

R914 *input-item* **is** *variable*
or *io-implied-do*

R915 *output-item* **is** *expr*
or *io-implied-do*

R916 *io-implied-do* **is** (*io-implied-do-object-list* , *io-implied-do-control*)

R917 *io-implied-do-object* **is** *input-item*
or *output-item*

R918 *io-implied-do-control* **is** *do-variable* = *scalar-int-expr* , ■
■ *scalar-int-expr* [, *scalar-int-expr*]

Constraint: A variable that is an *input-item* shall not be a whole assumed-size array.

Constraint: The *do-variable* shall be a named scalar variable of type integer.

Constraint: 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*.

An *input-item* shall not appear as, nor be associated with, the *do-variable* of any *io-implied-do* that contains the *input-item*.

If an input item is a pointer, it shall be currently 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 currently associated with a target and data are transferred from the target to the file.

NOTE 9.24

Data transfers always involve the movement of values between a file and internal storage. A pointer as such cannot be read or written. A pointer may, therefore, appear as an item in an input/output list if it is currently associated with a target that can receive a value (input) or can deliver a value (output). A derived-type object with one or more pointer components shall not appear as an item in an input/output list because the value of a pointer component is a descriptor for a location in memory. As such, this has no processor-independent representation.

If an input item or an output item is an allocatable array, it shall be currently allocated.

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*.

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 9.25

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
READ *, A (A (1) : A (10))                    ! Not allowed
```

A derived-type object shall not appear as an input/output list item if any component ultimately in the object is not accessible within the scoping unit containing the input/output statement.

NOTE 9.26

An example is a structure accessed from a module within which its type is PUBLIC but its components are PRIVATE.

If a derived type ultimately contains a pointer component, an object of this type shall not appear as an input item nor as the result of the evaluation of an output list item.

If a derived-type object appears as an input/output list item in a formatted input/output statement, it is treated as if all of the components of the object were specified in the same order as in the definition of the derived type.

NOTE 9.27

In a formatted input/output statement, edit descriptors are associated with effective list items, which are always scalar and of intrinsic type. The rules in 9.4.2 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 of intrinsic type.

An input/output list item of derived type in an unformatted input/output statement is treated as a single value in a processor-dependent form.

NOTE 9.28

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.

For an implied-DO, the loop initialization and execution is the same as for a DO construct (8.1.4.4).

An input/output list shall not contain an item of nondefault character type if the input/output statement specifies an internal file.

NOTE 9.29

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.

NOTE 9.30

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
```

9.4.3 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.

An **end-of-file condition** occurs in either of 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.

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.

If an error condition or an end-of-file condition occurs during execution of an input/output statement, execution of the input/output statement terminates and if the input/output statement contains any implied-DOs, all of the implied-DO variables in the statement become undefined. If an error condition occurs during execution of an input/output statement, the position of the file becomes indeterminate.

If an error or end-of-file condition occurs on input, all input list items become undefined.

If an end-of-record condition occurs during execution of a nonadvancing input statement, the following occurs: if the PAD= specifier has the value YES, the record is padded with blanks (9.4.4.4.2) to satisfy the input list item and corresponding data edit descriptor that require more characters than the record contains; execution of the input statement terminates and if the input statement contains any implied-DOs, all of the implied-DO variables in the statement become undefined; and the file specified in the input statement is positioned after the current record.

Execution of the program is terminated if an error condition occurs during execution of an input/output statement that contains neither an IOSTAT= nor an ERR= specifier, or if an end-of-file condition occurs during execution of a READ statement that contains neither an IOSTAT= specifier nor an END= specifier, or if an end-of-record condition occurs during execution of a nonadvancing READ statement that contains neither an IOSTAT= specifier nor an EOR= specifier.

9.4.4 Execution of a data transfer input/output statement

The effect of executing a 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 (9.2.1.3.2)

(5) Transfer data between the file and the entities specified by the input/output list (if any) or namelist

(6) Determine whether an error condition, an end-of-file condition, or an end-of-record condition has occurred

(7) Position the file after data transfer (9.2.1.3.3)

(8) Cause any variables specified in the IOSTAT= and SIZE= specifiers to become defined.

9.4.4.1 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. Execution of a WRITE or PRINT statement for a file that does not exist creates the file unless an error condition occurs.

9.4.4.2 Identifying a unit

A data transfer input/output statement that contains an input/output control list includes a unit specifier that identifies an external unit or an internal file. 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 unit or an internal file.

The unit identified by a data transfer input/output statement shall be connected to a file when execution of the statement begins.

NOTE 9.31

The file may be preconnected.

9.4.4.3 Establishing a format

If the input/output control list contains * as a format, list-directed formatting is established. If *namelist-group-name* is present, 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 format specifier is established. If the format is an array, the effect is as if all elements of the array were concatenated in array element order.

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.

9.4.4.4 Data transfer

Data are transferred between records and 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 next item to be processed in the list is called the **next effective item**. Zero-sized arrays and implied-DO lists with iteration counts of zero are ignored in determining the next effective item. A scalar character item of zero character length is treated as an effective item. 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 data objects (variables) are specified in the *namelist-group-object-list*.

All values needed to determine which entities are specified by an input/output list item are determined at the beginning of the processing of that item.

All values are transmitted to or from the entities specified by a list item prior to the processing of any succeeding list item for all data transfer input/output statements.

NOTE 9.32

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 (10.9) 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 9.33

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.

9.4.4.4.1 Unformatted data transfer

During unformatted data transfer, data are transferred without editing between the current record and the entities specified by the input/output list. Exactly one record is read or written.

Objects of intrinsic or derived types may be transferred by means of an unformatted data transfer statement.

On input, the file shall be positioned so that the record read is an unformatted record or an endfile record. The number of values required by the input list shall be less than or equal to the number of values in the record. Each value in the record shall be of the same type as the corresponding entity in the input list, except that one complex value may correspond to two real list entities or two real values may correspond to one complex list entity. The type parameters of the corresponding entities shall be the same.

NOTE 9.34

If an entity in the input list is of type character, the character entity shall have the same length and the same kind type parameter as the character value. Also, if two real values correspond to one complex entity or one complex value corresponds to two real entities, all three shall have the same kind type parameter value.

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 connected for formatted input/output, unformatted data transfer is prohibited.

The unit specified shall be an external unit.

9.4.4.4.2 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*, if any. 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.

On input, the file shall be positioned so that the record read is a formatted record or an endfile record.

If the file is connected for unformatted input/output, formatted data transfer is prohibited.

During advancing input from a file whose PAD= specifier 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 from a file whose PAD= specifier has the value YES, or during input from an internal file, blank characters are supplied by the processor if the input list and format specification require more characters from the record than the record contains.

During nonadvancing input from a file whose PAD= specifier has the value NO, an end-of-record condition (9.4.3) occurs if the input list and format specification require more characters from the record than the record contains.

During nonadvancing input from a file whose PAD= specifier has the value YES, an end-of-record condition occurs and blank characters are supplied by the processor if an input item and its corresponding data edit descriptor require more characters from the record than the record contains.

If the file is connected for direct access, the record number is increased by one as each succeeding record is read or written.

On output, if the file is connected for direct access or is an internal file and the characters specified by the output list and format do not fill a record, blank characters are added to fill the record.

On output, the output list and format specification shall not specify more characters for a record than have been specified by a RECL= specifier in the OPEN statement or the record length of an internal file.

9.4.4.5 List-directed formatting

If list-directed formatting has been established, editing is performed as described in 10.8.

9.4.4.6 Namelist formatting

If namelist formatting has been established, editing is performed as described in 10.9.

9.4.5 Printing of formatted records

The transfer of information in a formatted record to certain devices determined by the processor is called **printing**. If a formatted record is printed, the first character of the record is not printed. The remaining characters of the record, if any, are printed in one line beginning at the left margin.

The first character of such a record shall be of default character type and determines vertical spacing as follows:

Character	Vertical spacing before printing
Blank	One line
0	Two lines
1	To first line of next page
+	No advance

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

The PRINT statement does not imply that printing will occur, and the WRITE statement does not imply that printing will not occur.

9.4.6 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 (10.8, 10.9) 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 (9.4.3).

9.5 File positioning statements

R919 *backspace-stmt* **is** BACKSPACE *external-file-unit*
or BACKSPACE (*position-spec-list*)

R920 *endfile-stmt* **is** ENDFILE *external-file-unit*
or ENDFILE (*position-spec-list*)

R921 *rewind-stmt* **is** REWIND *external-file-unit*
or REWIND (*position-spec-list*)

A file that is not connected for sequential access shall not be referred to by a BACKSPACE, an ENDFILE, or a REWIND statement. A file that is connected with an ACTION= specifier having the value READ shall not be referred to by an ENDFILE statement.

R922 *position-spec* **is** [UNIT =] *external-file-unit*
or IOSTAT = *scalar-default-int-variable*
or ERR = *label*

Constraint: 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.

Constraint: If the optional characters UNIT= are omitted from the unit specifier, the unit specifier shall be the first item in the *position-spec-list*.

Constraint: A *position-spec-list* shall contain exactly one *external-file-unit* and may contain at most one of each of the other specifiers.

The IOSTAT= and ERR= specifiers are described in 9.4.1.4 and 9.4.1.5, respectively.

9.5.1 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 there is no current record and no preceding record, the position of the file is not changed.

NOTE 9.35
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 9.36

An example of a BACKSPACE statement is:

```
BACKSPACE (10, ERR = 20)
```

9.5.2 ENDFILE statement

Execution of an ENDFILE statement writes an endfile record as the next record of the file. The file is then positioned after the endfile record 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, 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 that is connected but does not exist creates the file prior to writing the endfile record.

NOTE 9.37

An example of an ENDFILE statement is:

```
ENDFILE K
```

9.5.3 REWIND statement

Execution of a REWIND statement causes the specified file to be positioned at its initial point.

NOTE 9.38

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.

NOTE 9.39

An example of a REWIND statement is:

```
REWIND 10
```

9.6 File inquiry

The INQUIRE statement may be used to inquire about properties of a particular named file or of the connection to a particular unit. There are 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.

R923 *inquire-stmt* **is** INQUIRE (*inquire-spec-list*)
or INQUIRE (IOLENGTH = *scalar-default-int-variable*) ■
■ *output-item-list*

NOTE 9.40

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)
```

NOTE 9.41

For more explanatory information on the INQUIRE statement, see C.6.5.

9.6.1 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:

R924 *inquire-spec* **is** [UNIT =] *external-file-unit*
or FILE = *file-name-expr*
or IOSTAT = *scalar-default-int-variable*
or ERR = *label*
or EXIST = *scalar-default-logical-variable*
or OPENED = *scalar-default-logical-variable*
or NUMBER = *scalar-default-int-variable*
or NAMED = *scalar-default-logical-variable*
or NAME = *scalar-default-char-variable*
or ACCESS = *scalar-default-char-variable*
or SEQUENTIAL = *scalar-default-char-variable*
or DIRECT = *scalar-default-char-variable*
or FORM = *scalar-default-char-variable*
or FORMATTED = *scalar-default-char-variable*
or UNFORMATTED = *scalar-default-char-variable*
or RECL = *scalar-default-int-variable*
or NEXTREC = *scalar-default-int-variable*
or BLANK = *scalar-default-char-variable*
or POSITION = *scalar-default-char-variable*
or ACTION = *scalar-default-char-variable*
or READ = *scalar-default-char-variable*
or WRITE = *scalar-default-char-variable*
or READWRITE = *scalar-default-char-variable*
or DELIM = *scalar-default-char-variable*
or PAD = *scalar-default-char-variable*

Constraint: An *inquire-spec-list* shall contain one FILE= specifier or one UNIT= specifier, but not both, and at most one of each of the other specifiers.

Constraint: In the inquire by unit form of the INQUIRE statement, if the optional characters UNIT= are omitted from the unit specifier, the unit specifier shall be the first item in the *inquire-spec-list*.

When a returned value of a specifier other than the NAME= specifier is of type character and the processor is capable of representing letters in both upper and lower case, the value returned is in upper case.

If an error condition occurs during execution of an INQUIRE statement, all of the inquiry specifier variables become undefined, except for the variable in the IOSTAT= specifier (if any).

The IOSTAT= and ERR= specifiers are described in 9.4.1.4 and 9.4.1.5, respectively.

9.6.1.1 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. If a processor is capable of representing letters in both upper and lower case, the interpretation of case is processor dependent.

9.6.1.2 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.

9.6.1.3 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.

9.6.1.4 NUMBER= specifier in the INQUIRE statement

The *scalar-default-int-variable* in the NUMBER= specifier is assigned the value of the external unit identifier of the unit that is currently connected to the file. If there is no unit connected to the file, the value –1 is assigned.

9.6.1.5 NAMED= specifier in the INQUIRE statement

The *scalar-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.

9.6.1.6 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 9.42

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.

If a processor is capable of representing letters in both upper and lower case, the case of the characters assigned to *scalar-default-char-variable* is processor dependent.

9.6.1.7 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, and DIRECT if the file is connected for direct access. If there is no connection, it is assigned the value UNDEFINED.

9.6.1.8 SEQUENTIAL= specifier in the INQUIRE statement

The *scalar-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.

9.6.1.9 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.

9.6.1.10 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.

9.6.1.11 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.

9.6.1.12 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.

9.6.1.13 RECL= specifier in the INQUIRE statement

The *scalar-default-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 processor-dependent units. If there is no connection, the *scalar-default-int-variable* becomes undefined.

9.6.1.14 NEXTREC= specifier in the INQUIRE statement

The *scalar-default-int-variable* in the NEXTREC= specifier is assigned the value $n + 1$, where n is the record number of the last record read or written on the file connected for direct access. If the file is connected but no records have been read or written since the connection, the *scalar-default-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-default-int-variable* becomes undefined.

9.6.1.15 BLANK= specifier in the INQUIRE statement

The *scalar-default-char-variable* in the BLANK= specifier is assigned the value NULL if null blank control is in effect for the file connected for formatted input/output, and is assigned the value ZERO if zero blank control is in effect for the file connected for formatted input/output. If there is no connection, or if the connection is not for formatted input/output, the *scalar-default-char-variable* is assigned the value UNDEFINED.

9.6.1.16 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.

9.6.1.17 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.

9.6.1.18 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.

9.6.1.19 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.

9.6.1.20 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.

9.6.1.21 DELIM= specifier in the INQUIRE statement

The *scalar-default-char-variable* in the DELIM= specifier is assigned the value APOSTROPHE if the apostrophe is to be used to delimit character data written by list-directed or namelist formatting. If the quotation mark is used to delimit such data, the value QUOTE is assigned. If neither the apostrophe nor the quote is used to delimit the character data, the value NONE is assigned. If there is no connection or if the connection is not for formatted input/output, the *scalar-default-char-variable* is assigned the value UNDEFINED.

9.6.1.22 PAD= specifier in the INQUIRE statement

The *scalar-default-char-variable* in the PAD= specifier is assigned the value NO if the connection of the file to the unit included the PAD= specifier and its value was NO. Otherwise, the *scalar-default-char-variable* is assigned the value YES.

9.6.2 Restrictions on inquiry specifiers

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.

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.

9.6.3 Inquire by output list

The inquire by output list form of the INQUIRE statement does not include a FILE= or UNIT= specifier, and includes only an IOLENGTH= specifier and an output list.

The *scalar-default-int-variable* in the IOLENGTH= specifier is assigned the processor-dependent value that would result from the use of the output list in an unformatted output statement. 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.

9.7 Restrictions on function references and list items

A function reference shall not appear in an expression anywhere in an input/output statement if such a reference causes another input/output statement to be executed.

NOTE 9.43

Restrictions in the evaluation of expressions (7.1.7) prohibit certain side effects.

9.8 Restriction on input/output statements

If a unit, or a file connected to a unit, does not have all of the properties required for the execution of certain input/output statements, those statements shall not refer to the unit.

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 format specifier (9.4.1.1) in an input/output statement may refer to a FORMAT statement or to a character expression that contains a format specification. A format specification provides explicit editing information. The format specifier also may be an asterisk (*) which indicates list-directed formatting (10.8). Instead of a format specifier, a *namelist-group-name* may be specified which indicates namelist formatting (10.9).

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*])

Constraint: The *format-stmt* shall be labeled.

Constraint: The comma used to separate *format-item*s 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 (10.6.5),

(2) Before a slash edit descriptor when the optional repeat specification is not present (10.6.2),

(3) After a slash edit descriptor, or

(4) Before or after a colon edit descriptor (10.6.3)

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 (10.7).

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 specifier 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 specifier references 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 specifier references 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 specifier 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*

Constraint: *r* shall be positive.

Constraint: *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*

R1006 *w* **is** *int-literal-constant*

R1007 *m* **is** *int-literal-constant*

R1008 *d* **is** *int-literal-constant*

R1009 *e* **is** *int-literal-constant*

Constraint: *e* shall be positive.

Constraint: *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.

Constraint: *w*, *m*, *d*, and *e* shall not have kind parameters specified for them.

I, B, O, Z, F, E, EN, ES, G, L, A, and D indicate the manner of editing.

R1010 *control-edit-desc* **is** *position-edit-desc*
or [*r*] **/**
or :
or *sign-edit-desc*
or *k* P
or *blank-interp-edit-desc*

R1011 *k* **is** *signed-int-literal-constant*

Constraint: *k* shall not have a kind parameter specified for it.

R1012 *position-edit-desc* **is** T *n*
or TL *n*
or TR *n*
or *n* X

R1013 *n* **is** *int-literal-constant*

Constraint: *n* shall be positive.

Constraint: *n* shall not have a kind parameter specified for it.

R1014 *sign-edit-desc* **is** S
or SP
or SS

R1015 *blank-interp-edit-desc* **is** BN
or BZ

In *k*P, *k* is called the **scale factor**.

T, TL, TR, X, slash, colon, S, SP, SS, P, BN, and BZ indicate the manner of editing.

R1016 *char-string-edit-desc* **is** *char-literal-constant*

Constraint: 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.

Within a character literal constant, appearances of the delimiter character itself, apostrophe or quote, shall be as consecutive pairs without intervening blanks. Each such pair represents a single occurrence of the delimiter character.

If a processor is capable of representing letters in both upper and lower case, 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 beginning of formatted data transfer using a format specification initiates **format control** (9.4.4.4.2). 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 (9.4.4.4). Zero length character items are effective list items, but zero sized arrays and implied-DO lists with an iteration count of zero are not.

Except for a format item preceded by a repeat specification *r*, a format specification is interpreted from left to right.

A format item preceded by a repeat specification is processed as a list of *r* items, each identical to the format item but without the repeat specification and separated by commas.

NOTE 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 (9.4.2), 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 (10.6.2). Format control then reverts to the beginning of the format item terminated by the last preceding right parenthesis. If there is no such preceding right parenthesis, format control reverts to the first left parenthesis of the format specification. If 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 scale factor (10.6.5.1), the sign control edit descriptors (10.6.4), or the blank interpretation edit descriptors (10.6.6).

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 10.6.1. After each slash edit descriptor is processed, the file is positioned as described in 10.6.2.

If format control reverts as described in 10.3, the file is positioned in a manner identical to the way it is positioned when a slash edit descriptor is processed (10.6.2).

During a read operation, any unprocessed characters of the current record are skipped whenever the next record is read.

10.5 Data edit descriptors

Data edit descriptors cause the conversion of data to or from its internal representation. Characters in the record shall be of default kind if they correspond to the value of a numeric, logical, or default character data entity, and shall be of nondefault kind if they correspond to the value of a data entity of nondefault character type. Characters transmitted to a record as a result of processing a character string edit descriptor shall be of default kind. 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.

10.5.1 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. The interpretation of blanks, other than leading blanks, is determined by a combination of any BLANK= specifier (9.3.4.6), the default for a preconnected or internal file, and any BN or BZ blank control that is currently in effect for the unit (10.6.6). Plus signs may be omitted. A field containing only blanks is considered to be zero.

(2) On input, with F, E, EN, ES, D, and G editing, a decimal point appearing in the input field overrides the portion of an edit descriptor that specifies the decimal point location. The input field may have more digits than the processor uses to approximate the value of the datum.

(3) On output 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, 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.

(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 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.

10.5.1.1 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 (10.5.4.1.1).

On input, *m* has no effect.

In the input field for the I edit descriptor, the character string shall be a *signed-digit-string* (R401), 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 (R408, R409, R410) in the respective input field. If a processor is capable of representing letters in both upper and lower case, 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 if the value of the internal datum is negative, or an optional plus 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 value of the internal datum 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 value of the internal datum is zero, one blank character is produced.

10.5.1.2 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 (10.5.4.1.2).

If a processor is capable of representing letters in both upper and lower case, a lower-case letter is equivalent to the corresponding upper-case letter in the exponent in a numeric input field.

10.5.1.2.1 F editing

The F*w.d* edit descriptor indicates that the field occupies *w* positions, the fractional part of which consists of *d* digits. When *w* is zero, the processor selects the field width. On input, *w* shall not be zero.

The input field consists of an optional sign, followed by a string of one or more digits optionally containing a decimal point, including any blanks interpreted as zeros. The *d* has no effect on input if the input field contains a decimal point. If the decimal point is omitted, the rightmost *d* digits of the string, with leading zeros assumed if necessary, are interpreted as the fractional part of the value represented. The string of digits may contain more digits than a processor uses to approximate the value of the constant. The basic form may be followed by an exponent of one of the following forms:

(1) 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 (10.6.5.1).

The output field consists of blanks, if necessary, followed by a minus if the internal value is negative, or an optional plus otherwise, followed by a string of digits that contains a decimal point and represents the magnitude of the internal value, as modified by the established scale factor and rounded to *d* fractional digits. Leading zeros are not permitted except for an optional zero immediately to the left of the decimal point if the magnitude of the value in the output field is less than one. The optional zero shall appear if there would otherwise be no digits in the output field.

10.5.1.2.2 E and D editing

The E*w.d*, D*w.d*, and E*w.d*E*e* edit descriptors indicate that the external field occupies *w* positions, the fractional part of which consists of *d* digits, unless a scale factor greater than one is in effect, and the exponent part consists of *e* digits. The *e* has no effect on input and *d* has no effect on input if the input field contains a decimal point.

The form and interpretation of the input field is the same as for F editing (10.5.1.2.1).

The form of the output field for a scale factor of zero is:

[±] [0].$x_1x_2\ldots x_d exp$

where:

± signifies a plus or a minus.

$x_1x_2\ldots x_d$ are the *d* most significant digits of the datum value after rounding.

exp is a decimal exponent having one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
E*w.d*	$\|exp\| \le 99$	$E\pm z_1z_2$ or $\pm 0z_1z_2$
	$99 < \|exp\| \le 999$	$\pm z_1z_2z_3$
E*w.d*E*e*	$\|exp\| \le 10^e - 1$	$E\pm z_1z_2\ldots z_e$

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
D*w.d*	$\|exp\| \le 99$	$D\pm z_1 z_2$ or $E\pm z_1 z_2$ or $\pm 0 z_1 z_2$
	$99 < \|exp\| \le 999$	$\pm z_1 z_2 z_3$

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 *Dw.d* shall not be used if $|exp| > 999$.

The scale factor k controls the decimal normalization (10.2.1, 10.6.5.1). If $-d < k \le 0$, the output field contains exactly $|k|$ leading zeros and $d - |k|$ significant digits after the decimal point. If $0 < k < d + 2$, the output field contains exactly k significant digits to the left of the decimal point and $d - k + 1$ significant digits to the right of the decimal point. Other values of k are not permitted.

10.5.1.2.3 EN editing

The EN edit descriptor 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 (R414) 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 editing (10.5.1.2.1).

The form of the output field is:

$[\ \pm\ ]\ yyy\ .\ x_1 x_2 \ldots x_d exp$

where:

$\pm$ signifies a plus or a minus.

yyy are the 1 to 3 decimal digits representative of the most significant digits of the value of the datum after rounding (yyy is an integer such that $1 \le yyy < 1000$ or, if the output value is zero, $yyy = 0$).

$x_1 x_2 \ldots x_d$ are the d next most significant digits of the value of the datum after rounding.

exp is a decimal exponent, divisible by three, of one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
EN*w.d*	$\|exp\| \le 99$	$E\pm z_1 z_2$ or $\pm 0 z_1 z_2$
	$99 < \|exp\| \le 999$	$\pm z_1 z_2 z_3$
EN*w.d*E*e*	$\|exp\| \le 10^e - 1$	$E\pm z_1 z_2 \ldots z_e$

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 EN*w.d* shall not be used if $|exp| > 999$.

NOTE 10.11

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

10.5.1.2.4 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 (R414) 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 editing (10.5.1.2.1).

The form of the output field is:

[±] y . $x_1x_2\ldots x_d exp$

where:

± signifies a plus or a minus.

y is a decimal digit representative of the most significant digit of the value of the datum after rounding.

$x_1x_2\ldots x_d$ are the *d* next most significant digits of the value of the datum 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*	$\|exp\| \le 99$	E$\pm z_1z_2$ or $\pm 0z_1z_2$
	$99 < \|exp\| \le 999$	$\pm z_1z_2z_3$
ES*w.d*E*e*	$\|exp\| \le 10^e - 1$	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 10.12

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

10.5.1.2.5 Complex editing

A complex datum consists of a pair of separate real data. The editing of a scalar datum of complex data 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.

10.5.2 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 (10.5.4.2).

The input field consists of optional blanks, optionally followed by a decimal point, followed by a T for true or F for false. The T or F may be followed by additional characters in the field, which are ignored.

If a processor is capable of representing letters in both upper and lower case, a lower-case letter is equivalent to the corresponding upper-case letter in a logical input field.

NOTE 10.13

The logical constants **.**TRUE**.** and **.**FALSE**.** are acceptable input forms.

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

10.5.3 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 (10.5.4.3). 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 A input is greater than or equal to *len*, the rightmost *len* characters will be taken from the input field. If the specified field width *w* is less than *len*, the *w* characters will appear left-justified with $len - w$ trailing blanks in the internal representation.

If the specified field width *w* for A output is greater than *len*, the output field will consist of $w - len$ blanks followed by the *len* characters from the internal representation. If the specified field width *w* is less than or equal to *len*, the output field will consist of the leftmost *w* characters from the internal representation.

NOTE 10.14

For nondefault character types, the blank padding character is processor dependent.

10.5.4 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.

10.5.4.1 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 (10.5.1).

NOTE 10.15

The G*w.d*E*e* edit descriptor follows any additional rules for the E*w.d*E*e* edit descriptor.

10.5.4.1.1 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 (10.5.1.1), except that *w* shall not be zero.

10.5.4.1.2 Generalized real and complex editing

The form and interpretation of the input field is the same as for F editing (10.5.1.2.1).

The method of representation in the output field depends on the magnitude of the datum being edited. Let N be the magnitude of the internal datum. If $0 < N < 0.1 - 0.5 \times 10^{-d-1}$ or $N \geq 10^{d} - 0.5$, 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 (10.6.5.1) currently in effect. If $0.1 - 0.5 \times 10^{-d-1} \leq N < 10^{d} - 0.5$ 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 Datum	Equivalent Conversion
$N = 0$	F(*w*–*n*).(*d*–1), *n*('*b*')
$0.1 - 0.5 \times 10^{-d-1} \leq N < 1 - 0.5 \times 10^{-d}$	F(*w*–*n*).*d*, *n*('*b*')
$1 - 0.5 \times 10^{-d} \leq N < 10 - 0.5 \times 10^{-d+1}$	F(*w*–*n*).(*d*–1), *n*('*b*')
$10 - 0.5 \times 10^{-d+1} \leq N < 100 - 0.5 \times 10^{-d+2}$	F(*w*–*n*).(*d*–2), *n*('*b*')
.	.
.	.
.	.
$10^{d-2} - 0.5 \times 10^{-2} \leq N < 10^{d-1} - 0.5 \times 10^{-1}$	F(*w*–*n*).1, *n*('*b*')
$10^{d-1} - 0.5 \times 10^{-1} \leq N < 10^{d} - 0.5$	F(*w*–*n*).1, *n*('*b*')

where *b* is a blank. *n* is 4 for G*w.d* and *e* + 2 for G*w.d*E*e*. *w* - *n* shall be positive.

NOTE 10.16

The scale factor has no effect unless the magnitude of the datum to be edited is outside the range that permits effective use of F editing.

10.5.4.2 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 (10.5.2).

10.5.4.3 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 (10.5.3).

10.6 Control edit descriptors

A control edit descriptor does not cause the transfer of data nor the conversion of data to or from internal representation, but may affect the conversions performed by subsequent data edit descriptors.

10.6.1 Position editing

The T, TL, TR, and X edit descriptors specify the position at which the next character will be transmitted to or from the record. If any character skipped by a T, TL, TR, or X edit descriptor is of type nondefault character, 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 10.17

An *n*X edit descriptor has the same effect as a TR*n* edit descriptor.

On output, a T, TL, TR, or X edit descriptor does not by itself cause characters to be transmitted and therefore does not by itself affect the length of the record. If characters are transmitted to positions at or after the position specified by a T, TL, TR, or X edit descriptor, positions skipped and not previously filled are filled with blanks. The result is as if the entire record were initially filled with blanks.

On output, a character in the record may be replaced. However, a T, TL, TR, or X edit descriptor never directly causes a character already placed in the record to be replaced. Such edit descriptors may result in positioning such that subsequent editing causes a replacement.

10.6.1.1 T, TL, and TR editing

The **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. 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.

NOTE 10.18

The T edit descriptor includes the vertical spacing character (9.4.5) in lines that are to be printed. T1 specifies the vertical spacing character and T2 specifies the first character that is printed. Only certain processor determined devices support vertical spacing characters.

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 10.19

The *n* in a T*n*, TL*n*, or TR*n* edit descriptor shall be specified and shall be greater than zero.

10.6.1.2 X editing

The *n*X edit descriptor indicates that the transmission of the next character to or from a record is to occur at the position *n* characters forward from the current position.

NOTE 10.20

The *n* in an *n*X edit descriptor shall be specified and shall be greater than zero.

10.6.2 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 access, the remaining portion of the current record is skipped and the file is positioned at the beginning of the next record. This record becomes the current record. On output to a file connected for sequential access, a new 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 10.21

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.

10.6.3 Colon editing

The colon edit descriptor terminates format control if there are no more effective items in the input/output list (9.4.2). The colon edit descriptor has no effect if there are more effective items in the input/output list.

10.6.4 S, SP, and SS editing

The S, SP, and SS edit descriptors may be used to control optional plus characters in numeric output fields. At the beginning of execution of each formatted output statement, the processor has the option of producing a plus in numeric output fields. If an SP edit descriptor is encountered in a format specification, the processor shall produce a plus in any subsequent position that normally contains an optional plus. If an SS edit descriptor is encountered, the processor shall not produce a plus in any subsequent position that normally contains an optional plus. If an S edit descriptor is encountered, the option of producing the plus is restored to the processor.

The S, SP, and SS edit descriptors affect only I, F, E, EN, ES, D, and G editing during the execution of an output statement. The S, SP, and SS edit descriptors have no effect during the execution of an input statement.

10.6.5 P editing

The *k*P edit descriptor redefines the scale factor to *k*. The scale factor may affect the editing of numeric quantities.

10.6.5.1 Scale factor

The value of the scale factor is zero at the beginning of execution of each input/output statement. It applies to all subsequently interpreted F, E, EN, ES, D, and G edit descriptors until a P edit descriptor is encountered, and then a new scale factor is established.

NOTE 10.22

Reversion of format control (10.3) does not affect the established scale factor.

The scale factor *k* affects the appropriate editing in the following manner:

(1) On input, with F, E, EN, 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 (R414) 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.

10.6.6 BN and BZ editing

The BN and BZ edit descriptors may be used to specify the interpretation of blanks, other than leading blanks, in numeric input fields. At the beginning of execution of each formatted input statement, nonleading blank characters from a file connected by an OPEN statement are interpreted as zeros or are ignored, depending on the value of the BLANK= specifier (9.3.4.6) currently in effect for the unit; an internal file or a preconnected file that has not been opened is treated as if the file had been opened with BLANK = 'NULL'. If a BN edit descriptor is encountered in a format specification, all nonleading blank characters in succeeding numeric input fields are ignored. The effect of ignoring blanks is to treat the input field as if blanks had been removed, the remaining portion of the field right-justified, and the blanks replaced as leading blanks. However, a field containing only blanks has the value zero. If a BZ edit descriptor is encountered in a format specification, all nonleading blank characters in succeeding numeric input fields are treated as zeros.

The BN and BZ edit descriptors affect only I, B, O, Z, F, E, EN, ES, D, and G editing during execution of an input statement. They have no effect during execution of an output statement.

10.7 Character string edit descriptors

A character string edit descriptor 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 10.23

A delimiter for a character string edit descriptor is either an apostrophe or quote.

10.8 List-directed formatting

The characters in one or more list-directed records constitute a sequence of values and value separators. The end of a record has the same effect as a blank character, unless it is within a character constant. Any sequence of two or more consecutive blanks is treated as a single blank, unless it is within a character constant.

Each value is either a null value or one of the forms:

c
*r***c*
*r**

where *c* is a literal constant or a nondelimited 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,

(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 10.24

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 10.25

List-directed input/output allows data editing according to the type of the list item instead of by a format specifier. It also allows data to be free-field, that is, separated by commas or blanks.

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.

10.8.1 List-directed input

Input forms acceptable to edit descriptors for a given type are acceptable for list-directed formatting, except as noted below. The form of the input value 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.

NOTE 10.26

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 (10.5.1.2.1) that is assumed to have no fractional digits unless a decimal point 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 comma, and followed by a right parenthesis. The first numeric input field is the real part of the complex constant and the second is the imaginary part. Each of the numeric input fields may be preceded or followed by

blanks. The end of a record may occur between the real part and the comma or between the comma and the imaginary part.

When the next effective item is of type logical, the input form shall not include slashes, blanks, or commas 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, and slash may appear in default character sequences.

If the next effective item is of type default character and

(1) The character sequence does not contain the value separators blank, comma, or slash,
(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 numeric 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 and 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 (7.5.1.4).

10.8.1.1 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 10.27

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 implied-DO variable in the input list is defined as though enough null values had been supplied for any remaining input list items.

NOTE 10.28

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

(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 10.29

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
```

10.8.2 List-directed output

The form of the values produced is the same as that required for input, except as noted otherwise. With the exception of adjacent nondelimited character sequences, the values are separated by one or more blanks or by a comma optionally preceded by one or more blanks and optionally followed by one or more blanks.

The processor may begin new records as necessary, but, except for complex constants and character sequences, the end of a record shall not occur within a constant or sequence and blanks shall not appear within a constant or sequence.

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 $10^{d_1} \le x < 10^{d_2}$, where d_1 and d_2 are processor-dependent integers. If the magnitude x is within this range, the constant is produced using 0PF*w.d*; otherwise, 1PE*w.d*E*e* is used.

For numeric output, reasonable processor-dependent values of *w*, *d*, and *e* are used for each of the numeric constants output.

Complex constants are enclosed in parentheses with a comma separating the real and imaginary parts, each produced as defined above for real constants. The end of a record may occur between

the comma and the imaginary part only if the entire constant is as long as, or longer than, an entire record. The only embedded blanks permitted within a complex constant are between the comma and the end of a record and one blank at the beginning of the next record.

Character sequences produced for an internal file, or for a file opened without a DELIM= specifier (9.3.4.9) or with a DELIM= specifier with a value of NONE

(1) Are not delimited by apostrophes or quotation marks,

(2) Are not 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 for carriage control at the beginning of any record that begins with the continuation of a character sequence from the preceding record.

Character sequences produced for a file opened with a DELIM= specifier with a value of QUOTE are delimited by quotes, are preceded and followed by a value separator, and have each internal quote represented on the external medium by two contiguous quotes.

Character sequences produced for a file opened with a DELIM= specifier with a value of APOSTROPHE are delimited by apostrophes, are preceded and followed by a value separator, and have each internal apostrophe represented on the external medium by two 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 to provide carriage control when the record is printed.

NOTE 10.30

The length of the output records is not specified exactly and may be processor dependent.

10.9 Namelist formatting

The characters in one or more namelist records constitute a sequence of **name-value subsequences**, each of which consists of an object name or a subobject 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 (10.9.1.4) or one of the forms

c
*r*c*
*r**

where *c* is a literal constant 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 (10.8).

10.9.1 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 10.31

A slash encountered in a namelist input record causes the input statement to terminate. A slash may not 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.

If a processor is capable of representing letters in both upper and lower case, a group name or object name is without regard to case.

10.9.1.1 Namelist group object names

Within the input data, each name shall correspond to a specific 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 subobject designators (R602). 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. The name in the input record may be preceded and followed by one or more optional blanks but shall not contain embedded blanks.

10.9.1.2 Namelist input values

The datum *c* (10.9) 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 10.9.1.3. The form of the 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 10.9.1.3.

The name-value subsequences are evaluated serially, in left-to-right order. A namelist group object name or subobject designator may appear in more than one name-value sequence.

When the name in the input record represents an array variable or a variable of derived type, the effect is as if the variable represented were expanded into a sequence of scalar list items of intrinsic data types, in the same way that formatted input/output list items are expanded (9.4.2). Each input value following the equals shall then be acceptable to format specifications for the intrinsic type of the list item in the corresponding position in the expanded sequence, except as noted in 10.9.1.3. The number of values following the equals shall not exceed the number of list items in the expanded sequence, but may be less; in the latter case, the effect is as if sufficient null values had been appended to match any remaining list items in the expanded sequence.

NOTE 10.32

For example, if the name in the input record is the name of an integer array of size 100, at most 100 values, each of which is either a digit string or a null value, may follow the equals; these values would then be assigned to the elements of the array in array element order.

A slash encountered as a value separator during the execution of a namelist input statement causes termination of execution of that input statement after assignment of the previous value. If there are additional items in the namelist group object being transferred, the effect is as if null values had been supplied for them.

A namelist comment may appear after any value separator except a slash. A namelist comment is also permitted to start in the first nonblank position of an input record except within a character literal constant.

Successive namelist records are read by namelist input until a slash is encountered; the remainder of the record is ignored and need not follow the rules for namelist input values.

10.9.1.3 Namelist group object list items

When the next effective namelist group object list item is of type real, the input form of the input value is that of a numeric input field. A numeric input field is a field suitable for F editing (10.5.1.2.1) that is assumed to have no fractional digits unless a decimal point appears within the field.

When the next effective item is of type complex, the input form of the input value consists of a left parenthesis followed by an ordered pair of numeric input fields separated by a comma and followed by a right parenthesis. The first numeric input field is the real part of the complex constant and the second part is the imaginary part. Each of the numeric input fields may be preceded or followed by blanks. The end of a record may occur between the real part and the comma or between the comma and the imaginary part.

When the next effective item is of type logical, the input form of the input value shall not include slashes, blanks, equals, or commas among the optional characters permitted for L editing (10.5.2).

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 character, the input form consists of a delimited sequence of zero or more *rep-chars* whose kind type parameter is implied by the kind of the corresponding list item. Such a sequence 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 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 10.33

A character sequence corresponding to a namelist input item of character data 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 (9.3.4.9).

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 (7.5.1.4).

10.9.1.4 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 10.34

The end of a record following a value separator, with or without intervening blanks, does not specify a null value in namelist input.

10.9.1.5 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, and

(4) Blanks between a name and the following equals.

10.9.1.6 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.

NOTE 10.35

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)
```

NOTE 10.35 *(Continued)*

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
```

10.9.2 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. If the processor is capable of representing letters in both upper and lower case, the name in the output is in upper case. With the exception of adjacent nondelimited character values, the values are separated by one or more blanks or by a 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 10.36

The length of the output records is not specified exactly and may be processor dependent.

10.9.2.1 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 $10^{d_1} \le x < 10^{d_2}$, where d_1 and d_2 are processor-dependent integers. If the magnitude x is within this range, the constant is produced using 0PF*w.d*; otherwise, 1PE*w.d*E*e* is used.

For numeric output, reasonable processor-dependent integer values of *w*, *d*, and *e* are used for each of the numeric constants output.

Complex constants are enclosed in parentheses with a comma separating the real and imaginary parts, each produced as defined above for real constants. The end of a record may occur between the comma and the imaginary part only if the entire constant is as long as, or longer than, an entire record. The only embedded blanks permitted within a complex constant are between the comma and the end of a record and one blank at the beginning of the next record.

Character sequences produced for a file opened without a DELIM= specifier (9.3.4.9) or with a DELIM= specifier with a value of NONE

(1) Are not delimited by apostrophes or quotation marks,

(2) Are not 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 for carriage control at the beginning of any record that begins with the continuation of a character sequence from the preceding record.

NOTE 10.37

Namelist output records produced with a DELIM= specifier with a value of NONE and which contain a character sequence may not be acceptable as namelist input records.

Character sequences produced for a file opened with a DELIM= specifier with a value of QUOTE are delimited by quotes, are preceded and followed by a value separator, and have each internal quote represented on the external medium by two contiguous quotes.

Character sequences produced for a file opened with a DELIM= specifier with a value of APOSTROPHE are delimited by apostrophes, are preceded and followed by a value separator, and have each internal apostrophe represented on the external medium by two contiguous apostrophes.

10.9.2.2 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 specific 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 to provide carriage control when the record is printed.

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 all of these program units except external subprograms, which are described in Section 12.

11.1 Main program

A **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*]]

Constraint: In a *main-program*, the *execution-part* shall not contain a RETURN statement or an ENTRY statement.

Constraint: 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*.

Constraint: An automatic object shall not appear in the *specification-part* (R204) of a main program.

The **program name** is global to the program, and shall not be the same as the name of any other program unit, external procedure, or common block in the program, nor the same as any local name in the main program.

NOTE 11.1

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 ANALYSE
   REAL A, B, C (10,10)     !  Specification part
   CALL FIND                !  Execution part
CONTAINS
   SUBROUTINE FIND          !  Internal subprogram
       ...
   END SUBROUTINE FIND
END PROGRAM ANALYSE
```

11.1.1 Main program specifications

The specifications in the scoping unit of the main program shall not include an OPTIONAL statement, an INTENT statement, a PUBLIC statement, a PRIVATE statement, or their equivalent attributes (5.1.2). A SAVE statement has no effect in a main program.

11.1.2 Main program executable part

The sequence of *execution-part* statements specifies the actions of the main program during program execution. Execution of a program (R201) begins with the first executable construct of the main program.

A main program shall not be recursive; that is, a reference to it shall not appear in any program unit in the program, including itself.

Normal execution of a program ends with execution of the *end-program-stmt* of the main program or with execution of a STOP statement in any program unit of the program. Execution may also be terminated if certain error conditions occur.

11.1.3 Main program internal subprograms

Any internal subprograms in the main program shall follow the CONTAINS statement. Internal subprograms are described in 12.1.2.2. The main program is called the **host** of its internal subprograms.

11.2 External subprograms

External subprograms are described in Section 12.

11.3 Modules

A **module** contains specifications and definitions that are to be accessible to other program units.

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*]]

Constraint: If the *module-name* is specified in the *end-module-stmt*, it shall be identical to the *module-name* specified in the *module-stmt*.

Constraint: A module *specification-part* shall not contain a *stmt-function-stmt*, an *entry-stmt*, or a *format-stmt*.

Constraint: An automatic object shall not appear in the *specification-part* (R204) of a module.

Constraint: If an object of a type for which *component-initialization* is specified (R429) appears in the *specification-part* of a module and does not have the ALLOCATABLE or POINTER attribute, the object shall have the SAVE attribute.

The module name is global to the program, and shall not be the same as the name of any other program unit, external procedure, or common block in the program, nor be the same as any local name in the module.

NOTE 11.3

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 11.4

For a discussion of the impact of modules on dependent compilation, see section C.8.2.

NOTE 11.5

For examples of the use of modules, see section C.8.3.

11.3.1 Module reference

A USE statement specifying a module name is a **module reference**. At the time a module reference is processed, the public portions of the specified module shall be available. A module shall not reference itself, either directly or indirectly.

The accessibility, public or private, of specifications and definitions in a module to a scoping unit making reference to the module may be controlled in both the module and the scoping unit making the reference. In the module, the PRIVATE statement, the PUBLIC statement (5.2.3), their equivalent attributes (5.1.2.2), and the PRIVATE statement in a derived-type definition (4.4.1) are used to control the accessibility of module entities outside the module.

NOTE 11.6

For a discussion of the impact of accessibility on dependent compilation, see section C.8.2.2.

In a scoping unit making reference to a module, the ONLY option in the USE statement may be used to further limit the accessibility, in that referencing scoping unit, of the public entities in the module.

11.3.2 The USE statement and use association

The **USE statement** provides the means by which a scoping unit accesses named data objects, derived types, interface blocks, procedures, 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.

R1107 *use-stmt* **is** USE *module-name* [, *rename-list*]
or USE *module-name* , ONLY : [*only-list*]

R1108 *rename* **is** *local-name* => *use-name*

R1109 *only* **is** *generic-spec*
or *only-use-name*
or *only-rename*

R1110 *only-use-name* **is** *use-name*

R1111 *only-rename* **is** *local-name* => *use-name*

Constraint: Each *generic-spec* shall be a public entity in the module.

Constraint: Each *use-name* shall be the name of a public entity in the module.

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 or *use-name*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 named in one or more of the *only-list*s are accessible.

An accessible entity in the referenced module has one or more local names. These names are

(1) The name of the entity in the referenced module if that name appears as an *only-use-name* in any *only* for that module,

(2) Each of the *local-name*s the entity is given in any *rename* or *only-rename* for that module, and

(3) The name of the entity in the referenced module if that name does not appear as a *use-name* in any *rename* or *only-rename* for that module.

Two or more accessible entities, other than generic interfaces, may have the same name only if the name is not used to refer to an entity in the scoping unit. Generic interfaces are handled as described in section 14.1.2.3. Except for these cases, the local name of any entity given accessibility by a USE statement shall differ from the local names of all other entities accessible to the scoping unit through USE statements and otherwise.

NOTE 11.7

There is no prohibition against a *use-name* 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 name.

The local name of an entity made accessible by a USE statement may appear in no other specification statement that would cause any attribute (5.1.2) of the entity to be respecified 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.

NOTE 11.8

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-object*.

The appearance of such a local name 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 name appears in a PRIVATE statement in a module, the entity is not a public entity of that module. If the local name does not appear in either a PUBLIC or PRIVATE statement, it assumes the default accessibility attribute (5.2.3) of that scoping unit.

A procedure with an implicit interface and public accessibility shall explicitly be given the EXTERNAL attribute in the scoping unit of the module; if it is a function, its type and type parameters shall be explicitly declared in a type declaration statement in that scoping unit.

An intrinsic procedure with public accessibility shall explicitly be given the INTRINSIC attribute in the scoping unit of the module or be used as an intrinsic procedure in that scoping unit.

NOTE 11.9

For a discussion of the impact of the ONLY clause and renaming on dependent compilation, see section C.8.2.1.

NOTE 11.10

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.

NOTE 11.10 *(Continued)*

```
USE STATS_LIB, ONLY : YPROD; USE STATS_LIB
```

makes all public entities in STATS_LIB accessible.

11.4 Block data program units

A **block data program unit** is used to provide initial values for data objects in named common blocks.

R1112 *block-data* **is** *block-data-stmt*
[*specification-part*]
end-block-data-stmt

R1113 *block-data-stmt* **is** BLOCK DATA [*block-data-name*]

R1114 *end-block-data-stmt* **is** END [BLOCK DATA [*block-data-name*]]

Constraint: The *block-data-name* may be included in the *end-block-data-stmt* only if it was provided in the *block-data-stmt* and, if included, shall be identical to the *block-data-name* in the *block-data-stmt*.

Constraint: A *block-data specification-part* may contain only USE statements, type declaration statements, IMPLICIT statements, PARAMETER statements, derived-type definitions, and the following specification statements: COMMON, DATA, DIMENSION, EQUIVALENCE, INTRINSIC, POINTER, SAVE, and TARGET.

Constraint: A type declaration statement in a *block-data specification-part* shall not contain ALLOCATABLE, EXTERNAL, INTENT, OPTIONAL, PRIVATE, or PUBLIC attribute specifiers.

NOTE 11.11

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 11.12

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 11.13

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 11.14

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 reference may identify, as actual arguments, entities that are associated during execution of the procedure reference with corresponding dummy arguments in the procedure definition.

12.1 Procedure classifications

A procedure is classified according to the form of its reference and the way it is defined.

12.1.1 Procedure classification by reference

The definition of a procedure specifies it to be a function or a subroutine. A reference to a function 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 (7.5.1.3).

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, 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 entity, an internal subprogram shall not contain an ENTRY statement, the internal procedure name shall not be argument associated with a dummy procedure (12.4.1.2), 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.

If a subprogram contains one or more ENTRY statements, it defines a procedure for each ENTRY statement and a procedure for the SUBROUTINE or FUNCTION statement.

12.1.2.3 Dummy procedures

A dummy argument that is specified as a procedure or appears in a procedure reference is a **dummy procedure**.

12.1.2.4 Statement functions

A function that is defined by a single statement is a **statement function** (12.5.4).

12.2 Characteristics of procedures

The **characteristics of a procedure** are the classification of the procedure as a function or subroutine, whether or not it is pure, whether or not it is elemental, 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 is either a dummy data object, a dummy procedure, 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 (5.1.2.3, 5.2.1), whether it is optional (5.1.2.6, 5.2.2), and whether it is a pointer (5.1.2.7, 5.2.7) or a target (5.1.2.8, 5.2.8). If a type parameter of an object or a bound of an array is an expression that depends on the value or attributes of another object, the exact dependence on other entities is a characteristic. If a shape, size, or character length is assumed, it is a characteristic.

12.2.1.2 Characteristics of dummy procedures

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, and whether it is optional (5.1.2.6, 5.2.2).

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, and whether it is a pointer. If a function result is an array that is not 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 a constant expression, the exact dependence on the entities in the expression 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 interface consists of the characteristics of the procedure, the name of the procedure, the name and characteristics of each dummy argument, 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.

NOTE 12.1

For more explanatory information on procedure interfaces, see section 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 recursive subroutine or a recursive 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 block (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

(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) An optional dummy argument,
  (b) A dummy argument that is an assumed-shape array, a pointer, or a target,
  (c) An array-valued result (functions only),
  (d) A result that is a pointer (functions only), or
  (e) A result whose character length parameter value is not assumed and not constant (character functions only), or

(3) The procedure is elemental.

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 block, or in both except that the ENTRY statement shall not appear in an interface block.

NOTE 12.3

An interface block 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 module procedure may appear in a MODULE PROCEDURE statement in an interface block (12.3.2.1).

12.3.2.1 Procedure interface block

R1201 *interface-block* **is** *interface-stmt*
  [*interface-specification*] ...
  end-interface-stmt

R1202 *interface-specification* **is** *interface-body*
  or *module-procedure-stmt*

R1203 *interface-stmt* **is** INTERFACE [*generic-spec*]

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

Constraint: An *interface-body* of a pure procedure shall specify the intents of all dummy arguments except pointer, alternate return, and procedure arguments.

R1206 *module-procedure-stmt* **is** MODULE PROCEDURE *procedure-name-list*

R1207 *generic-spec* **is** *generic-name*
or OPERATOR (*defined-operator*)
or ASSIGNMENT (=)

Constraint: An *interface-body* shall not contain an *entry-stmt*, *data-stmt*, *format-stmt*, or *stmt-function-stmt*.

Constraint: The MODULE PROCEDURE statement is allowed only if the *interface-block* has a *generic-spec* and is in a scoping unit where each *procedure-name* is accessible as a module procedure.

Constraint: An *interface-block* in a subprogram shall not contain an *interface-body* for a procedure defined by that subprogram.

Constraint: The *generic-spec* may be included in the *end-interface-stmt* only if it was provided in the *interface-stmt* and, if included, shall be identical to the *generic-spec* in the *interface-stmt*.

Constraint: A *procedure-name* in a *module-procedure-stmt* shall not be one which previously had been specified in any *module-procedure-stmt* with the same generic identifier in the same specification part.

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 body** in an interface block specifies an explicit specific interface for an existing external procedure or a dummy procedure. If the name in a procedure heading in an interface block is the same as the name of a dummy argument in the subprogram containing the interface block, the interface block declares that dummy argument to be a dummy procedure with the indicated interface; otherwise, the interface block declares the name to be the name of an external procedure with the indicated procedure interface.

An interface body specifies all of the procedure's characteristics and these 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. 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. An interface block shall not contain an ENTRY statement, but an entry interface may be specified by using the entry name as the procedure name in the interface body. 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.

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
```

NOTE 12.5 *(Continued)*

```
   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)
```

An interface block with a generic specification specifies a **generic interface** for each of the procedures in the interface block. If the interface block is in a module or is in a scoping unit that accesses a module by use association, the MODULE PROCEDURE statement lists those module procedures that have this generic interface. The listed module procedures may be defined in the module containing the interface block or may be accessible via a USE statement. The characteristics of module procedures are not given in interface blocks, but are assumed from the module subprograms. A generic interface is always explicit.

Any procedure may be referenced via its specific interface. It also may be referenced via its generic interface, if it has one. The generic name, defined operator, or equals symbol in a generic specification is a **generic identifier** for all the procedures in the interface block. The rules on how any two procedures with the same generic identifier shall differ are given in 14.1.2.3. 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.

NOTE 12.6

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:

```
CALL SWITCH (MAX_VAL, LOC_VAL) ! MAX_VAL and LOC_VAL are of type INTEGER
```

12.3.2.1.1 Defined operations

If OPERATOR is specified in a generic specification, all of the procedures specified in the interface block shall be functions that may be referenced as defined operations (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) and 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.

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 12.7

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 (7.3), extending one form (such as <=) has the effect of defining both forms (<= and .LE.).

12.3.2.1.2 Defined assignments

If ASSIGNMENT is specified in an INTERFACE statement, all the procedures in the interface block shall be subroutines that may be referenced as defined assignments (7.5.1.3). 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). 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 the assignment operation is extended, or redefined if both sides of the equals sign are of the same derived type.

NOTE 12.8

An example of the use of the ASSIGNMENT generic specification is:

```
INTERFACE ASSIGNMENT ( = )

   SUBROUTINE LOGICAL_TO_NUMERIC  (N, B)
      INTEGER, INTENT (OUT) :: N
```

NOTE 12.8 *(Continued)*

```
      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'))
```

12.3.2.2 EXTERNAL statement

An **EXTERNAL statement** specifies the EXTERNAL attribute for a list of names.

R1208 *external-stmt* **is** EXTERNAL [::] *external-name-list*

Each *external-name* shall be the name of an external procedure, a dummy argument, or a block data program unit.

The appearance of the name of a dummy argument in an EXTERNAL statement specifies that the dummy argument is a dummy procedure. The appearance in an EXTERNAL statement of a name that is not the name of a dummy argument specifies that the name is the name of an external procedure or block data program unit.

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. Appearance of an intrinsic procedure name in an EXTERNAL statement causes that name to become the name of an external procedure and thus the intrinsic procedure of the same name is not available in the scoping unit.

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.

Only one appearance of a name in all of the EXTERNAL statements in a scoping unit is permitted. A name that appears in an EXTERNAL statement in a given scoping unit or is a use-associated entity with the EXTERNAL attribute shall not also appear as a specific procedure name in an interface block in the scoping unit nor in an interface block that is accessible to the scoping unit.

NOTE 12.9

For explanatory information on potential portability problems with external procedures, see section C.9.1.

NOTE 12.10

An example of an EXTERNAL statement is:

```
EXTERNAL FOCUS
```

12.3.2.3 INTRINSIC statement

An **INTRINSIC statement** specifies a list of names that have the INTRINSIC attribute. A name that has the INTRINSIC attribute represents an intrinsic procedure (Section 13). The INTRINSIC attribute permits a name that represents a specific intrinsic function to be used as an actual argument.

R1209 *intrinsic-stmt* **is** INTRINSIC [::] *intrinsic-procedure-name-list*

Constraint: Each *intrinsic-procedure-name* shall be the name of an intrinsic procedure.

The appearance of a name in an INTRINSIC statement confirms that the name is the name of an intrinsic procedure. The appearance of a generic intrinsic procedure name (13.11, 13.12) in an INTRINSIC statement does not cause that name to lose its generic property. In a scoping unit, a name may appear both as the name of a generic intrinsic procedure in an INTRINSIC statement and as the name of a generic interface, provided that the procedures in the interface and the specific intrinsic procedures are all functions or all subroutines (14.1.2.3).

If the specific name of an intrinsic function (13.13) is used as an actual argument, the name shall either appear in an INTRINSIC statement or be given the INTRINSIC attribute in a type declaration statement in the scoping unit.

Only one appearance of a name in all of the INTRINSIC statements in a scoping unit is permitted.

NOTE 12.11

A name shall not appear in both an EXTERNAL and an INTRINSIC statement in the same scoping unit.

12.3.2.4 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, but is independent of the means by which the procedure is defined. The forms of procedure references are:

R1210 *function-reference* **is** *function-name* ([*actual-arg-spec-list*])

Constraint: The *actual-arg-spec-list* for a function reference shall not contain an *alt-return-spec*.

R1211 *call-stmt* **is** CALL *subroutine-name* [([*actual-arg-spec-list*])]

A function may also be referenced as a defined operation. A subroutine may also be referenced as a defined assignment.

R1212 *actual-arg-spec* **is** [*keyword* =] *actual-arg*

R1213 *keyword* **is** *dummy-arg-name*

R1214 *actual-arg* **is** *expr*
or *variable*
or *procedure-name*
or *alt-return-spec*

R1215 *alt-return-spec* **is** * *label*

Constraint: The *keyword* = shall not appear if the interface of the procedure is implicit in the scoping unit.

Constraint: The *keyword* = may be omitted from an *actual-arg-spec* only if the *keyword* = has been omitted from each preceding *actual-arg-spec* in the argument list.

Constraint: Each *keyword* shall be the name of a dummy argument in the explicit interface of the procedure.

Constraint: A non-intrinsic elemental procedure shall not be used as an actual argument.

Constraint: A *procedure-name actual-arg* shall not be the name of an internal procedure or of a statement function and shall not be the generic name of a procedure unless it is also a specific name (12.3.2.1, 13.1).

NOTE 12.12

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 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.

Constraint: In a reference to a pure procedure, a *procedure-name actual-arg* shall be the name of a pure procedure (12.6).

NOTE 12.13

This constraint ensures that the purity of a procedure cannot be undermined by allowing it to call a nonpure procedure.

Constraint: 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 12.14

Successive commas shall not be used to omit optional arguments.

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. In the absence of an argument keyword, an actual argument is associated with the dummy argument occupying the corresponding position in the dummy argument list; that is, the first actual argument is associated with the first dummy argument, the second actual argument is associated with the second dummy argument, etc. If an argument keyword is present, 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). 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 12.15

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 Actual arguments associated with dummy data objects

If a dummy argument is a dummy data object, the associated actual argument shall be an expression of the same type or a data object of the same type. The kind type parameter value of

the actual argument shall agree with that of the dummy argument. The value of the character length parameter of an actual argument of type nondefault character shall agree with that of the dummy argument. If the dummy argument is an assumed-shape array of type default character, the value of the character length parameter of the actual argument shall agree with that of the dummy argument. If the dummy argument is an assumed-shape array, the rank of the dummy argument shall agree with the rank of the actual argument.

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, the restriction on length is for the entire array and not for each array element. The length of an array element in the dummy argument array may be different from the length of an array element in the associated actual argument array, array element, or array element substring, but the dummy argument array shall not extend beyond the end of the actual argument array.

Except when a procedure reference is elemental (12.7), each element of an array-valued actual argument or of a sequence in a sequence association (12.4.1.4) is associated with the element of the dummy array that has the same position in array element order (6.2.2.2).

NOTE 12.16

For type default character sequence associations, the interpretation of element is provided in 12.4.1.4.

If the dummy argument is a pointer, the actual argument shall be a pointer and the types, type parameters, and ranks shall agree.

At the invocation of the procedure, the dummy argument pointer receives the pointer association status of the actual argument. If the actual argument is currently associated, the dummy argument becomes associated with the same target. The association status may change during the execution of the procedure. When execution of the procedure completes, the pointer association status of the dummy argument becomes undefined if it is associated with a target that becomes undefined (14.7.6); following this, the pointer association status of the actual argument becomes that of the dummy argument.

If the dummy argument is not a pointer and the corresponding actual argument is a pointer, the actual argument shall be currently associated with a target and the dummy argument becomes argument associated with that target.

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 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 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

(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 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

(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 actual argument is scalar, the corresponding dummy argument shall be scalar unless the actual argument is an element of an array that is not an assumed-shape or pointer array, or a substring of such an element. 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 actual argument shall not be an assumed-size array or a scalar (including an array element designator or an array element substring designator).

A scalar dummy argument of a nonelemental procedure may be associated only with a scalar actual argument.

If a 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.

If the actual argument is an array section having a vector subscript, the dummy argument is not definable and shall not have INTENT (OUT) or INTENT (INOUT).

NOTE 12.17

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.

NOTE 12.18

For more explanatory information on argument association and evaluation, see section C.9.4. For more explanatory information on pointers and targets as dummy arguments, see section C.9.5.

12.4.1.2 Actual arguments associated with dummy procedures

If a dummy argument is a dummy procedure, the associated actual argument shall be the specific name of an external, module, dummy, or intrinsic procedure. The only intrinsic procedures permitted are those listed in 13.13 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 the interface of the dummy procedure is explicit, the characteristics listed in 12.2 shall be the same for the associated actual procedure and the corresponding dummy procedure, except that a pure actual procedure may be associated with a dummy procedure that is not pure and an elemental intrinsic actual procedure may be associated with a dummy procedure that is not elemental.

If the interface of the dummy procedure is implicit and either the name of the dummy procedure is explicitly typed or the procedure is referenced as a function, the dummy procedure shall not be referenced as a subroutine and the actual argument shall be a function or dummy procedure.

If the interface of the dummy procedure is implicit and a reference to the procedure appears as a subroutine reference, the actual argument shall be a subroutine or dummy procedure.

12.4.1.3 Actual arguments associated with alternate return indicators

If a dummy argument is an asterisk (12.5.2.3), the associated actual argument shall be an alternate return specifier. The label in the alternate return specifier shall identify an executable construct in the scoping unit containing the procedure reference.

12.4.1.4 Sequence association

An actual argument represents an **element sequence** if it is an array expression, an array element designator, or an array element substring designator. 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 and is an array expression, array element, or array element substring designator, the element sequence consists of the character storage units beginning with the first storage unit of the actual argument and continuing to the end of the array. The character storage units of an array element substring designator are viewed as array elements consisting of consecutive groups of character storage units having the character length of the dummy array.

NOTE 12.19

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-valued data object 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.

12.4.1.5 Restrictions on dummy arguments not present

A dummy argument is **present** in an instance of a subprogram if it is associated with an actual argument and the actual argument either is a dummy argument that is present in the invoking subprogram or is not a dummy argument of the invoking subprogram. 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 dummy 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) If it is a dummy procedure, it shall not be invoked.

(3) 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.

(4) A subobject of it shall not be supplied as an actual argument corresponding to an optional dummy argument.

(5) 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.

(6) If it is a pointer, it shall not be supplied as an actual argument corresponding to a nonpointer dummy argument other than as the argument of the PRESENT intrinsic function.

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.

12.4.1.6 Restrictions on entities associated with dummy arguments

While an entity is associated with a dummy argument, the following restrictions hold:

(1) No action that affects the allocation status of the entity may be taken. Action that affects the value of the entity or any part of it shall be taken through the dummy argument unless

   (a) the dummy argument has the POINTER attribute,

   (b) the part is all or part of a pointer subobject, or

   (c) 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 12.20

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
```

NOTE 12.20 *(Continued)*

an assignment statement such as

```
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 12.21

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 12.22

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)
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 12.23

Since a dummy argument declared with an intent of 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 value of any part of the entity 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,

(b) the part is all or part of a pointer subobject, or

(c) 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 12.24

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 12.25

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 or defined assignment statement (7.5.1.3). When a subroutine is invoked, all actual argument expressions are evaluated, then the arguments are associated, and then the subroutine is executed. When the actions specified by the subroutine are completed, execution of the CALL statement or defined assignment statement is also completed. If a CALL statement includes one or more alternate return specifiers among its arguments, control may be transferred to one of the statements indicated, depending on the action specified by the subroutine. A reference to an elemental subroutine (12.7) is an elemental reference if all actual arguments corresponding to INTENT (OUT) and INTENT (INOUT) dummy arguments are arrays that have the same shape and the remaining actual arguments are conformable with them.

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 (12.5.2.4) 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 or with the END statement if there is no other executable construct.

12.5.2.1 Effects of INTENT attribute on subprograms

The INTENT attribute of dummy data objects limits the way in which they may be used in a subprogram. A dummy data object having INTENT (IN) shall neither be defined nor become undefined during the execution of the procedure. A dummy data object having INTENT (OUT) is initially undefined in the subprogram unless the object is of a type for which default initialization is specified. A dummy data object with INTENT (INOUT) may be referenced or be defined. A dummy data object whose intent is not specified is subject to the limitations of the data entity that is the associated actual argument. That is, a reference to the dummy data object may occur if the actual argument is defined and the dummy data object may be defined if the actual argument is definable.

12.5.2.2 Function subprogram

A **function subprogram** is a subprogram that has a FUNCTION statement as its first statement.

R1216 *function-subprogram* **is** *function-stmt*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-function-stmt

R1217 *function-stmt* **is** [*prefix*] FUNCTION *function-name* ■
■ ([*dummy-arg-name-list*]) [RESULT (*result-name*)]

Constraint: If RESULT is specified, the *function-name* shall not appear in any specification statement in the scoping unit of the function subprogram.

R1218 *prefix* **is** *prefix-spec* [*prefix-spec*] ...

R1219 *prefix-spec* **is** *type-spec*
or RECURSIVE
or PURE
or ELEMENTAL

Constraint: A *prefix* shall contain at most one of each *prefix-spec*.

Constraint: If ELEMENTAL is present, RECURSIVE shall not be present.

R1220 *end-function-stmt* **is** END [FUNCTION [*function-name*]]

Constraint: If RESULT is specified, *result-name* shall not be the same as *function-name*.

Constraint: FUNCTION shall be present in the *end-function-stmt* of an internal or module function.

Constraint: An internal function subprogram shall not contain an ENTRY statement.

Constraint: An internal function subprogram shall not contain an *internal-subprogram-part*.

Constraint: If a *function-name* is present in the *end-function-stmt*, it shall be identical to the *function-name* specified in the *function-stmt*.

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 statement in the declaration part of the function subprogram. It shall not be specified both ways. If it is not specified either way, it is determined by the implicit typing rules in force within the function subprogram. If the function result is array-valued or 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 be present if the function directly or indirectly invokes itself or a function defined by an ENTRY statement in the same subprogram. Similarly, RECURSIVE shall be present 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.

The name of the function is *function-name*.

If RESULT is specified, the name of the result variable of the function is *result-name*, its characteristics (12.2.2) are those of the function result, and all occurrences of the function name in *execution-part* statements in the scoping unit are recursive function references. 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 value of the result variable at the completion of execution of the function is the value returned by the function. If the function result has been declared to be 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 has been declared 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 12.26

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 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 is present, the subprogram is a pure subprogram and shall meet the additional constraints of 12.6.

If the *prefix-spec* ELEMENTAL is present, the subprogram is an elemental subprogram and shall meet the additional constraints of 12.7.1.

If both RECURSIVE and RESULT are specified, the interface of the function being defined is explicit within the function subprogram.

NOTE 12.27

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 .LE. 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
```

12.5.2.3 Subroutine subprogram

A **subroutine subprogram** is a subprogram that has a SUBROUTINE statement as its first statement.

R1221 *subroutine-subprogram* **is** *subroutine-stmt*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-subroutine-stmt

R1222 *subroutine-stmt* **is** [*prefix*] SUBROUTINE *subroutine-name* ■
■ [([*dummy-arg-list*])]

Constraint: The *prefix* of a *subroutine-stmt* shall not contain a *type-spec*.

R1223 *dummy-arg* **is** *dummy-arg-name*
or *

R1224 *end-subroutine-stmt* **is** END [SUBROUTINE [*subroutine-name*]]

Constraint: SUBROUTINE shall be present in the *end-subroutine-stmt* of an internal or module subroutine.

Constraint: An internal subroutine subprogram shall not contain an ENTRY statement.

Constraint: An internal subroutine subprogram shall not contain an *internal-subprogram-part*.

Constraint: If a *subroutine-name* is present in the *end-subroutine-stmt*, it shall be identical to the *subroutine-name* specified in the *subroutine-stmt*.

The *prefix-spec* RECURSIVE shall be present if the subroutine directly or indirectly invokes itself or a subroutine defined by an ENTRY statement in the same subprogram. Similarly, RECURSIVE shall be present 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 RECURSIVE is specified, the interface of the subroutine being defined is explicit within the subroutine subprogram.

The name of the subroutine is *subroutine-name*.

If the *prefix-spec* PURE or ELEMENTAL is present, the subprogram is a pure subprogram and shall meet the additional constraints of 12.6.

If the *prefix-spec* ELEMENTAL is present, the subprogram is an elemental subprogram and shall meet the additional constraints of 12.7.1.

12.5.2.4 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 nonsaved 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 12.28

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.

12.5.2.5 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.

R1225 *entry-stmt* **is** ENTRY *entry-name* [([*dummy-arg-list*]) ■
■ [RESULT (*result-name*)]]

Constraint: 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.

Constraint: An *entry-stmt* may appear only in an *external-subprogram* or *module-subprogram*. An *entry-stmt* shall not appear within an *executable-construct*.

Constraint: RESULT may be present only if the *entry-stmt* is in a function subprogram.

Constraint: 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 and it shall not appear in an EXTERNAL or INTRINSIC statement.

Constraint: A *dummy-arg* may be an alternate return indicator only if the ENTRY statement is in a subroutine subprogram.

Constraint: If RESULT is specified, *result-name* shall not be the same as *entry-name*.

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 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 scalars without the POINTER

attribute and one of the types: default integer, default real, double precision real, default complex, or default logical.

If RESULT is specified in the ENTRY statement and RECURSIVE is specified in the FUNCTION statement, the interface of the function defined by the ENTRY statement is explicit within the function subprogram.

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.

If RECURSIVE is specified in the SUBROUTINE statement, the interface of the subroutine defined by the ENTRY statement is explicit within the subroutine subprogram.

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 program.

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 (12.4.1.5).

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.

12.5.2.6 RETURN statement

R1226 *return-stmt* **is** RETURN [*scalar-int-expr*]

Constraint: The *return-stmt* shall be in the scoping unit of a function or subroutine subprogram.

Constraint: 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 is present and has a value *n* between 1 and the number of asterisks in the dummy argument list, the CALL statement that invoked the subroutine transfers control to the statement identified by the *n*th alternate return specifier in the actual argument list. If the expression is omitted or has a value outside the required range, there is no transfer of control to an alternate return.

Execution of an *end-function-stmt* or *end-subroutine-stmt* is equivalent to executing a RETURN statement with no expression.

12.5.2.7 CONTAINS statement

R1227 *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. The CONTAINS statement is not executable.

12.5.3 Definition of procedures by means other than Fortran

The means other than Fortran by which a procedure may be defined are processor dependent. A reference to such a procedure is made as though it were defined by an external subprogram. The definition of a non-Fortran procedure shall not be in a Fortran program unit and a Fortran program unit shall not be in the definition of a non-Fortran procedure. The interface to a non-Fortran procedure may be specified in an interface block.

NOTE 12.29

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.

R1228 *stmt-function-stmt* **is** *function-name* ([*dummy-arg-name-list*]) = *scalar-expr*

Constraint: 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 array valued, 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.

Constraint: 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 parent array shall have been declared as an array earlier in the scoping unit or made accessible by use or host association.

Constraint: 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.

Constraint: The *function-name* and each *dummy-arg-name* shall be specified, explicitly or implicitly, to be scalar.

Constraint: A given *dummy-arg-name* may appear only once in any *dummy-arg-name-list*.

Constraint: 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 (13.10),

(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 the syntax rules defining non-intrinsic pure function subprograms (R1216-R1220) or non-intrinsic pure subroutine subprograms (R1221-R1224).

Constraint: The *specification-part* of a pure function subprogram shall specify that all dummy arguments have INTENT (IN) except procedure arguments and arguments with the POINTER attribute.

Constraint: The *specification-part* of a pure subroutine subprogram shall specify the intents of all dummy arguments except procedure arguments, alternate return indicators, and arguments with the POINTER attribute.

Constraint: A local variable declared in the *specification-part* or *internal-subprogram-part* of a pure subprogram shall not have the SAVE attribute.

NOTE 12.30

Variable initialization in a *type-declaration-stmt* or a *data-stmt* implies the SAVE attribute; therefore, such initialization is also disallowed.

Constraint: The *specification-part* of a pure subprogram shall specify that all dummy arguments that are procedure arguments are pure.

Constraint: 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.

NOTE 12.31

It is expected that most mathematical library procedures will be pure. This form of restriction allows these procedures to be used in contexts where they are not required to be pure without the need for an *interface-block*.

Constraint: All internal subprograms in a pure subprogram shall be pure.

Constraint: In a pure subprogram any variable which is in common or accessed by host or use association, is a dummy argument to a pure function, is a dummy argument with INTENT (IN) to a pure subroutine, or an object that is storage associated with any such variable, shall not be used in the following contexts:

(1) As the *variable* of an *assignment-stmt*;

(2) As a DO variable or implied DO variable;

(3) As an *input-item* in a *read-stmt* from an internal file;

(4) As an *internal-file-unit* in a *write-stmt*;

(5) As an IOSTAT= specifier in an input or output statement with an internal file;

(6) As the *pointer-object* of a *pointer-assignment-stmt*;

(7) As the *target* of a *pointer-assignment-stmt*;

(8) As the *expr* of an *assignment-stmt* in which the *variable* is of a derived type if the derived type has a pointer component at any level of component selection;

NOTE 12.32

This requires that processors be able to determine if entities with the PRIVATE attribute or with private components have a pointer component.

(9) As an *allocate-object* or *stat-variable* in an *allocate-stmt* or *deallocate-stmt*, or as a *pointer-object* in a *nullify-stmt*; or

(10) As an actual argument associated with a dummy argument with INTENT (OUT) or INTENT (INOUT) or with the POINTER attribute.

Constraint: Any procedure referenced in a pure subprogram, including one referenced via a defined operation or assignment, shall be pure.

Constraint: A pure subprogram shall not contain a *print-stmt*, *open-stmt*, *close-stmt*, *backspace-stmt*, *endfile-stmt*, *rewind-stmt*, or *inquire-stmt*.

Constraint: A pure subprogram shall not contain a *read-stmt* or *write-stmt* whose *io-unit* is an *external-file-unit* or *.

Constraint: A pure subprogram shall not contain a *stop-stmt*.

NOTE 12.33

The above constraints are designed to guarantee that a pure procedure is free from side effects (i.e., 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 I/O 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. It is also anticipated that most library procedures will not reference global data. Referencing global data may inhibit concurrent execution.

NOTE 12.34

Pure subroutines are included to allow subroutine calls from pure procedures in a safe way, and to allow *forall-assignment*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 be present; it is implied by the ELEMENTAL *prefix-spec*. The following additional constraints apply to the syntax rules defining elemental function subprograms (R1216-R1220) or elemental subroutine subprograms (R1221-R1224).

Constraint: All dummy arguments shall be scalar and shall not have the POINTER attribute.

Constraint: For a function, the result shall be scalar and shall not have the POINTER attribute.

Constraint: A dummy argument, or a subobject thereof, 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 (13.11.8).

Constraint: A *dummy-arg* shall not be *.

Constraint: A *dummy-arg* shall not be a dummy procedure.

NOTE 12.35

An elemental subprogram is a pure subprogram and all of the constraints for pure subprograms also apply.

Note 12.36

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 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-valued case, the values of the elements, if any, of the result are the same as would have been obtained if the scalar-valued function had been applied separately, in any order, to corresponding elements of each array actual argument.

NOTE 12.37

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. Apart from this, the actual arguments in a reference to an elemental subroutine must satisfy the restrictions of 12.4.1.6.

Section 13: Intrinsic procedures

There are four classes of intrinsic procedures: inquiry functions, elemental functions, transformational functions, and subroutines. One intrinsic subroutine is elemental.

13.1 Intrinsic functions

An **intrinsic function** is an inquiry function, an elemental intrinsic function, or a transformational function. An **inquiry function** is one whose result depends on the properties of its principal argument other than the value of this argument; in fact, the argument value may be undefined. 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-valued arguments or an array-valued result. All intrinsic functions defined in this standard are pure.

NOTE 13.1

Intrinsic subroutines are used for functionalities involving side effects.

Generic names of intrinsic functions are listed in 13.11. In most cases, generic functions accept arguments of more than one type and the type of the result is the same as the type of the arguments. **Specific names** of intrinsic functions with corresponding generic names are listed in 13.13.

If an intrinsic function is used as an actual argument to a procedure, its specific name shall be used and it may be referenced in the called procedure only with scalar arguments. If an intrinsic function does not have a specific name, it shall not be used as an actual argument (12.4.1.2).

13.2 Elemental intrinsic procedures

Elemental intrinsic procedures behave as described in 12.7.

13.3 Arguments to intrinsic procedures

All intrinsic procedures may be invoked with either positional arguments or argument keywords (12.4). The descriptions in 13.11 through 13.14 give the argument keyword names and positional sequence.

Many of the intrinsic procedures have optional arguments. These arguments are identified by the notation "optional" in the argument descriptions. In addition, the names of the optional arguments are enclosed in square brackets in description headings and in lists of procedures. The valid forms of reference for procedures with optional arguments are described in 12.4.1.

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 13.13 have INTENT(IN). The nonpointer dummy arguments of the generic intrinsic procedures in 13.14 have INTENT(IN) if the intent is not stated explicitly.

The actual argument associated with an intrinsic function dummy argument called KIND shall be a scalar integer initialization expression and shall specify a representation method for the function result that exists on the processor.

13.4 Argument presence inquiry function

The inquiry function PRESENT permits an inquiry to be made about the presence of an actual argument associated with a dummy argument that has the OPTIONAL attribute.

13.5 Numeric, mathematical, character, kind, logical, and bit procedures

13.5.1 Numeric functions

The elemental functions INT, REAL, DBLE, and CMPLX perform type conversions. The elemental functions AIMAG, CONJG, AINT, ANINT, NINT, ABS, MOD, SIGN, DIM, DPROD, MODULO, FLOOR, CEILING, MAX, and MIN perform simple numeric operations.

13.5.2 Mathematical functions

The elemental functions SQRT, EXP, LOG, LOG10, SIN, COS, TAN, ASIN, ACOS, ATAN, ATAN2, SINH, COSH, and TANH evaluate elementary mathematical functions.

13.5.3 Character functions

The elemental functions ICHAR, CHAR, LGE, LGT, LLE, LLT, IACHAR, ACHAR, INDEX, VERIFY, ADJUSTL, ADJUSTR, SCAN, and LEN_TRIM perform character operations. The transformational function REPEAT returns repeated concatenations of a character string argument. The transformational function TRIM returns the argument with trailing blanks removed.

13.5.4 Character inquiry function

The inquiry function LEN returns the length of a character entity. If the argument to this function consists of a single primary (7.1.1.1) that is a variable name, the variable need not be defined. It is not necessary for a processor to evaluate the argument of this function if the value of the function can be determined otherwise.

13.5.5 Kind functions

The inquiry function KIND returns the kind type parameter value of an integer, real, complex, logical, or character entity. The value of the argument to this function need not be defined. The transformational function SELECTED_REAL_KIND returns the real kind type parameter value that has at least the decimal precision and exponent range specified by its arguments. The transformational function SELECTED_INT_KIND returns the integer kind type parameter value that has at least the decimal exponent range specified by its argument.

13.5.6 Logical function

The elemental function LOGICAL converts between objects of type logical with different kind type parameter values.

13.5.7 Bit manipulation and inquiry procedures

The bit manipulation procedures consist of a set of 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 13.7.1 would have $z = 32$, thereby defining a 32-bit integer.

An inquiry function BIT_SIZE is available to determine the parameter z of the model. If the argument to this function consists of a single primary (7.1.1.1) that is a variable name, the variable need not be defined, if a pointer it may have undefined or disassociated status, and if allocatable it need not be allocated. It is not necessary for a processor to evaluate the argument of this function if the value of the function can be determined otherwise.

Effectively, this model defines an integer object to consist of z bits in sequence numbered from right to left from 0 to $z-1$. This model is valid only in the context of the use of such an object as the argument or result of one of the bit manipulation procedures. In all other contexts, the model defined for an integer in 13.7.1 applies. In particular, whereas the models are identical for $w_{z-1} = 0$, they do not correspond for $w_{z-1} = 1$ and the interpretation of bits in such objects is processor dependent.

13.6 Transfer function

The transformational function TRANSFER specifies that the physical representation of the first argument is to be treated as if it were one of the type and type parameters of the second argument with no conversion.

13.7 Numeric manipulation and inquiry functions

The numeric manipulation and inquiry functions are described in terms of a model for the representation and behavior of numbers on a processor. The model has parameters which are determined so as to make the model best fit the machine on which the program is executed.

13.7.1 Models for integer and real data

The model set for integer i is defined by

$$i = s \times \sum_{k=0}^{q-1} w_k \times r^k,$$

where r is an integer exceeding one, q is a positive integer, each w_k is a nonnegative integer less than r, and s is +1 or –1.

The model set for real x is defined by

$$: = \begin{cases} 0 \quad \text{or} \\ s \times b^{e} \times \sum_{k=1}^{p} f_k \times b^{-k}, \end{cases}$$

where b and p are integers exceeding one; each f_k is a nonnegative integer less than b, with f_1 nonzero; s is +1 or –1; and e is an integer that lies between some integer maximum $e_{\max}$ and some integer minimum $e_{\min}$ inclusively. For $x = 0$, its exponent e and digits f_k are defined to be zero. The integer parameters r and q determine the set of model integers and the integer parameters b, p, $e_{\min}$, and $e_{\max}$ determine the set of model floating point numbers. The parameters of the integer and real models are available for each integer and real data type implemented by the processor. The parameters characterize the set of available numbers in the definition of the model. The numeric manipulation and inquiry functions provide values related to the parameters and other constants related to them.

NOTE 13.4

Examples of these functions in this section use the models

$$i = s \times \sum_{k=0}^{30} w_k \times 2^{k}$$

and

$$x = 0 \text{ or } s \times 2^{e} \times \left(\frac{1}{2} + \sum_{k=2}^{24} f_k \times 2^{-k} \right), \quad -126 \le e \le 127$$

13.7.2 Numeric inquiry functions

The inquiry functions RADIX, DIGITS, MINEXPONENT, MAXEXPONENT, PRECISION, RANGE, HUGE, TINY, and EPSILON return scalar values related to the parameters of the model associated with the types and kind type parameters of the arguments. If the argument to these functions consists of a single primary (7.1.1.1) that is a variable name, the variable need not be defined, if a pointer it may have undefined or disassociated association status, and if allocatable it need not be allocated.

13.7.3 Floating point manipulation functions

The elemental functions EXPONENT, SCALE, NEAREST, FRACTION, SET_EXPONENT, SPACING, and RRSPACING return values related to the components of the model values (13.7.1) associated with the actual values of the arguments.

13.8 Array intrinsic functions

The array intrinsic functions perform the following operations on arrays: vector and matrix multiplication, numeric or logical computation that reduces the rank, array structure inquiry, array construction, array manipulation, and geometric location.

13.8.1 The shape of array arguments

The transformational array intrinsic functions operate on each array argument as a whole. The shape of the corresponding actual argument shall therefore be defined; that is, the actual argument shall be an array section, an assumed-shape array, an explicit-shape array, a pointer that is associated with a target, an allocatable array that has been allocated, or an array-valued expression. It shall not be an assumed-size array.

Some of the inquiry intrinsic functions accept array arguments for which the shape need not be defined. Assumed-size arrays may be used as arguments to these functions; they include the function LBOUND and certain references to SIZE and UBOUND.

13.8.2 Mask arguments

Some array intrinsic functions have an optional MASK argument that is used by the function to select the elements of one or more arguments to be operated on by the function. Any element not selected by the mask need not be defined at 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.

A MASK argument shall be of type logical.

13.8.3 Vector and matrix multiplication functions

The matrix multiplication function MATMUL operates on two matrices, or on one matrix and one vector, and returns the corresponding matrix-matrix, matrix-vector, or vector-matrix product. The arguments to MATMUL may be numeric (integer, real, or complex) or logical arrays. On logical matrices and vectors, MATMUL performs Boolean matrix multiplication.

The dot product function DOT_PRODUCT operates on two vectors and returns their scalar product. The vectors are of the same type (numeric or logical) as for MATMUL. For logical vectors, DOT_PRODUCT returns the Boolean scalar product.

13.8.4 Array reduction functions

The array reduction functions SUM, PRODUCT, MAXVAL, MINVAL, COUNT, ANY, and ALL perform numerical, logical, and counting operations on arrays. They may be applied to the whole array to give a scalar result or they may be applied over a given dimension to yield a result of rank reduced by one. By use of a logical mask that is conformable with the given array, the computation may be confined to any subset of the array (for example, the positive elements).

13.8.5 Array inquiry functions

The function ALLOCATED returns a value true if the array argument is currently allocated, and returns false otherwise. The functions SIZE, SHAPE, LBOUND, and UBOUND return, respectively, the size of the array, the shape, and the lower and upper bounds of the subscripts along each dimension. The size, shape, or bounds shall be defined.

If an argument to these functions consists of a single primary (7.1.1.1) that is a variable name, the variable need not be defined.

13.8.6 Array construction functions

The functions MERGE, SPREAD, PACK, and UNPACK construct new arrays from the elements of existing arrays. MERGE combines two conformable arrays into one array by an element-wise choice based on a logical mask. SPREAD constructs an array from several copies of an actual argument (SPREAD does this by adding an extra dimension, as in forming a book from copies of one page). PACK and UNPACK respectively gather and scatter the elements of a one-dimensional array from and to positions in another array where the positions are specified by a logical mask.

13.8.7 Array reshape function

RESHAPE produces an array with the same elements and a different shape.

13.8.8 Array manipulation functions

The functions TRANSPOSE, EOSHIFT, and CSHIFT manipulate arrays. TRANSPOSE performs the matrix transpose operation on a two-dimensional array. The shift functions leave the shape of an array unaltered but shift the positions of the elements parallel to a specified dimension of the array. These shifts are either circular (CSHIFT), in which case elements shifted off one end reappear at the other end, or end-off (EOSHIFT), in which case specified boundary elements are shifted into the vacated positions.

13.8.9 Array location functions

The functions MAXLOC and MINLOC return the location (subscripts) of an element of an array that has a maximum and minimum value, respectively. By use of an optional logical mask that is conformable with the given array, the reduction may be confined to any subset of the array.

13.9 Pointer association status functions

The transformational function NULL returns a disassociated pointer. The inquiry function ASSOCIATED tests whether a pointer is currently associated with any target, with a particular target, or with the same target as another pointer.

13.10 Intrinsic subroutines

Intrinsic subroutines are supplied by the processor and are defined in 13.12 and 13.14. An intrinsic subroutine is referenced by a CALL statement that uses its name explicitly. An intrinsic subroutine shall not be used as an actual argument. The effect of a subroutine reference is as specified in 13.14. The elemental subroutine MVBITS is pure. No other intrinsic subroutine defined in this standard is pure.

NOTE 13.5

As with user-written elemental subroutines, an elemental intrinsic subroutine is pure. The non-elemental intrinsic subroutines all have side effects (or reflect system side effects) and thus are not pure.

13.10.1 Date and time subroutines

The subroutines DATE_AND_TIME and SYSTEM_CLOCK return data from the date and real-time clock. The time returned is local, but there are facilities for finding out the difference between local time and Coordinated Universal Time. The subroutine CPU_TIME returns the processor time consumed during execution.

13.10.2 Pseudorandom numbers

The subroutine RANDOM_NUMBER returns a pseudorandom number or an array of pseudorandom numbers. The subroutine RANDOM_SEED initializes or restarts the pseudorandom number sequence.

13.10.3 Bit copy subroutine

The elemental subroutine MVBITS copies a bit field from a specified position in one integer object to a specified position in another.

13.11 Generic intrinsic functions

For all of the intrinsic procedures, the arguments shown are the names that shall be used for argument keywords when using the keyword form for actual arguments.

NOTE 13.6

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 13.7

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	Indicates a string scan is to be from right to left (backward)
MASK	A mask that may be applied to the arguments
DIM	A selected dimension of an array argument

13.11.1 Argument presence inquiry function

PRESENT (A)	Argument presence

13.11.2 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

13.11.3 Mathematical functions

ACOS (X)	Arccosine
ASIN (X)	Arcsine
ATAN (X)	Arctangent
ATAN2 (Y, X)	Arctangent
COS (X)	Cosine
COSH (X)	Hyperbolic cosine
EXP (X)	Exponential

LOG (X)	Natural logarithm
LOG10 (X)	Common logarithm (base 10)
SIN (X)	Sine
SINH (X)	Hyperbolic sine
SQRT (X)	Square root
TAN (X)	Tangent
TANH (X)	Hyperbolic tangent

13.11.4 Character functions

ACHAR (I)	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)	Position of a character in ASCII collating sequence
ICHAR (C)	Position of a character in processor collating sequence
INDEX (STRING, SUBSTRING [, BACK])	Starting position of a substring
LEN_TRIM (STRING)	Length without trailing blank characters
LGE (STRING_A, STRING_B)	Lexically greater than or equal
LGT (STRING_A, STRING_B)	Lexically greater than
LLE (STRING_A, STRING_B)	Lexically less than or equal
LLT (STRING_A, STRING_B)	Lexically less than
REPEAT (STRING, NCOPIES)	Repeated concatenation
SCAN (STRING, SET [, BACK])	Scan a string for a character in a set
TRIM (STRING)	Remove trailing blank characters
VERIFY (STRING, SET [, BACK])	Verify the set of characters in a string

13.11.5 Character inquiry function

LEN (STRING)	Length of a character entity

13.11.6 Kind functions

KIND (X)	Kind type parameter value
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

13.11.7 Logical function

LOGICAL (L [, KIND])	Convert between objects of type logical with different kind type parameters

13.11.8 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

13.11.9 Bit inquiry function

BIT_SIZE (I)	Number of bits of the model

13.11.10 Bit manipulation functions

BTEST (I, POS)	Bit testing
IAND (I, J)	Logical 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
NOT (I)	Logical complement

13.11.11 Transfer function

TRANSFER (SOURCE, MOLD [, SIZE])	Treat first argument as if of type of second argument

13.11.12 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

13.11.13 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

13.11.14 Array reduction functions

ALL (MASK [, DIM])	True if all values are true
ANY (MASK [, DIM])	True if any value is true
COUNT (MASK [, DIM])	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

13.11.15 Array inquiry functions

ALLOCATED (ARRAY)	Array allocation status
LBOUND (ARRAY [, DIM])	Lower dimension bounds of an array
SHAPE (SOURCE)	Shape of an array or scalar
SIZE (ARRAY [, DIM])	Total number of elements in an array
UBOUND (ARRAY [, DIM])	Upper dimension bounds of an array

13.11.16 Array construction functions

MERGE (TSOURCE, FSOURCE, MASK)	Merge under mask
PACK (ARRAY, MASK [, VECTOR])	Pack an array into an array of rank one under a mask
SPREAD (SOURCE, DIM, NCOPIES)	Replicates array by adding a dimension
UNPACK (VECTOR, MASK, FIELD)	Unpack an array of rank one into an array under a mask

13.11.17 Array reshape function

RESHAPE (SOURCE, SHAPE[, PAD, ORDER])	Reshape an array

13.11.18 Array manipulation functions

CSHIFT (ARRAY, SHIFT [, DIM])	Circular shift
EOSHIFT (ARRAY, SHIFT [, BOUNDARY, DIM])	End-off shift
TRANSPOSE (MATRIX)	Transpose of an array of rank two

13.11.19 Array location functions

MAXLOC (ARRAY, DIM [, MASK]) or MAXLOC (ARRAY [, MASK])	Location of a maximum value in an array
MINLOC (ARRAY, DIM [, MASK]) or MINLOC (ARRAY [, MASK])	Location of a minimum value in an array

13.11.20 Pointer association status functions

ASSOCIATED (POINTER [, TARGET])	Association status inquiry or comparison
NULL ([MOLD])	Returns disassociated pointer

13.12 Intrinsic subroutines

CPU_TIME (TIME)	Obtain processor time
DATE_AND_TIME ([DATE, TIME, ZONE, VALUES])	Obtain date and time
MVBITS (FROM, FROMPOS, LEN, TO, TOPOS)	Copies bits from one integer to another
RANDOM_NUMBER (HARVEST)	Returns pseudorandom number
RANDOM_SEED ([SIZE, PUT, GET])	Initializes or restarts the pseudorandom number generator
SYSTEM_CLOCK ([COUNT, COUNT_RATE, COUNT_MAX])	Obtain data from the system clock

13.13 Specific names for intrinsic functions

	Specific Name	Generic Name	Argument Type
	ABS (A)	ABS (A)	default real
	ACOS (X)	ACOS (X)	default real
	AIMAG (Z)	AIMAG (Z)	default complex
	AINT (A)	AINT (A)	default real
	ALOG (X)	LOG (X)	default real
	ALOG10 (X)	LOG10 (X)	default real
•	AMAX0 (A1, A2 [, A3,...])	REAL (MAX (A1, A2 [, A3,...]))	default integer
•	AMAX1 (A1, A2 [, A3,...])	MAX (A1, A2 [, A3,...])	default real
•	AMIN0 (A1, A2 [, A3,...])	REAL (MIN (A1, A2 [, A3,...]))	default integer
•	AMIN1 (A1, A2 [, A3,...])	MIN (A1, A2 [, A3,...])	default real
	AMOD (A, P)	MOD (A, P)	default real
	ANINT (A)	ANINT (A)	default real
	ASIN (X)	ASIN (X)	default real
	ATAN (X)	ATAN (X)	default real
	ATAN2 (Y, X)	ATAN2 (Y, X)	default real
	CABS (A)	ABS (A)	default complex
	CCOS (X)	COS (X)	default complex
	CEXP (X)	EXP (X)	default complex
•	CHAR (I)	CHAR (I)	default integer
	CLOG (X)	LOG (X)	default complex
	CONJG (Z)	CONJG (Z)	default complex
	COS (X)	COS (X)	default real
	COSH (X)	COSH (X)	default real
	CSIN (X)	SIN (X)	default complex
	CSQRT (X)	SQRT (X)	default complex
	DABS (A)	ABS (A)	double precision real
	DACOS (X)	ACOS (X)	double precision real
	DASIN (X)	ASIN (X)	double precision real
	DATAN (X)	ATAN (X)	double precision real
	DATAN2 (Y, X)	ATAN2 (Y, X)	double precision real
	DCOS (X)	COS (X)	double precision real
	DCOSH (X)	COSH (X)	double precision real
	DDIM (X, Y)	DIM (X, Y)	double precision real
	DEXP (X)	EXP (X)	double precision real
	DIM (X, Y)	DIM (X, Y)	default real
	DINT (A)	AINT (A)	double precision real
	DLOG (X)	LOG (X)	double precision real
	DLOG10 (X)	LOG10 (X)	double precision real
•	DMAX1 (A1, A2 [, A3,...])	MAX (A1, A2 [, A3,...])	double precision real
•	DMIN1 (A1, A2 [, A3,...])	MIN (A1, A2 [, A3,...])	double precision real
	DMOD (A, P)	MOD (A, P)	double precision real
	DNINT (A)	ANINT (A)	double precision real
	DPROD (X, Y)	DPROD (X, Y)	default real
	DSIGN (A, B)	SIGN (A, B)	double precision real
	DSIN (X)	SIN (X)	double precision real
	DSINH (X)	SINH (X)	double precision real
	DSQRT (X)	SQRT (X)	double precision real
	DTAN (X)	TAN (X)	double precision real
	DTANH (X)	TANH (X)	double precision real
	EXP (X)	EXP (X)	default real
•	FLOAT (A)	REAL (A)	default integer

IABS (A)	ABS (A)	default integer
• ICHAR (C)	ICHAR (C)	default character
IDIM (X, Y)	DIM (X, Y)	default integer
• IDINT (A)	INT (A)	double precision real
IDNINT (A)	NINT (A)	double precision real
• IFIX (A)	INT (A)	default real
INDEX (STRING, SUBSTRING)	INDEX (STRING, SUBSTRING)	default character
• INT (A)	INT (A)	default real
ISIGN (A, B)	SIGN (A, B)	default integer
LEN (STRING)	LEN (STRING)	default character
• LGE (STRING_A, STRING_B)	LGE (STRING_A, STRING_B)	default character
• LGT (STRING_A, STRING_B)	LGT (STRING_A, STRING_B)	default character
• LLE (STRING_A, STRING_B)	LLE (STRING_A, STRING_B)	default character
• LLT (STRING_A, STRING_B)	LLT (STRING_A, STRING_B)	default character
• MAX0 (A1, A2 [, A3,...])	MAX (A1, A2 [, A3,...])	default integer
• MAX1 (A1, A2 [, A3,...])	INT (MAX (A1, A2 [, A3,...]))	default real
• MIN0 (A1, A2 [, A3,...])	MIN (A1, A2 [, A3,...])	default integer
• MIN1 (A1, A2 [, A3,...])	INT (MIN (A1, A2 [, A3,...]))	default real
MOD (A, P)	MOD (A, P)	default integer
NINT (A)	NINT (A)	default real
• REAL (A)	REAL (A)	default integer
SIGN (A, B)	SIGN (A, B)	default real
SIN (X)	SIN (X)	default real
SINH (X)	SINH (X)	default real
• SNGL (A)	REAL (A)	double precision real
SQRT (X)	SQRT (X)	default real
TAN (X)	TAN (X)	default real
TANH (X)	TANH (X)	default real

• These specific intrinsic functions shall not be used as an actual argument.

13.14 Specifications of the intrinsic procedures

This section contains detailed specifications of the generic intrinsic procedures in alphabetical order.

The types and type parameters of intrinsic procedure arguments and function results are determined by these specifications. 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.

13.14.1 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).

13.14.2 ACHAR (I)

Description. Returns the character in a specified position of the ASCII collating sequence. It is the inverse of the IACHAR function.

Class. Elemental function.

Argument. I shall be of type integer.

Result Characteristics. Character of length one with kind type parameter value KIND ('A').

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; otherwise, the result is processor dependent. If the processor is not capable of representing both upper- and lower-case letters and I corresponds to a letter in a case that the processor is not capable of representing, the result is the letter in the case that the processor is capable of representing. ACHAR (IACHAR (C)) shall have the value C for any character C capable of representation in the processor.

Example. ACHAR (88) has the value 'X'.

13.14.3 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 $0 \leq \mathrm{ACOS}(X) \leq \pi$.

Example. ACOS (0.54030231) has the value 1.0 (approximately).

13.14.4 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 '.

13.14.5 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'.

13.14.6 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 value y.

Example. AIMAG ((2.0, 3.0)) has the value 3.0.

13.14.7 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 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.

13.14.8 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 $1 \leq \mathrm{DIM} \leq n$, 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 or MASK has rank one; otherwise, the result is an array 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 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) has a value equal to that of ALL (MASK). Otherwise, the value of element $(s_1, s_2, ..., s_{\mathrm{DIM}-1}, s_{\mathrm{DIM}+1}, ..., s_n)$ of ALL (MASK, DIM) is equal to ALL (MASK $(s_1, s_2, ..., s_{\mathrm{DIM}-1}, :, s_{\mathrm{DIM}+1}, ..., s_n)$).

Examples.

Case (i): The value of ALL ((/ .TRUE., .FALSE., .TRUE. /)) is false.

Case (ii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ and C is the array $\begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$ then ALL (B .NE. C, DIM = 1) is [true, false, false] and ALL (B .NE. C, DIM = 2) is [false, false].

13.14.9 ALLOCATED (ARRAY)

Description. Indicate whether or not an allocatable array is currently allocated.

Class. Inquiry function.

Argument. ARRAY shall be an allocatable array.

Result Characteristics. Default logical scalar.

Result Value. The result has the value true if ARRAY is currently allocated and has the value false if ARRAY is not currently allocated.

13.14.10 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 KIND; otherwise, the kind type parameter is that of A.

Result Value. If A > 0, ANINT (A) has the value AINT (A + 0.5); if $A \leq 0$, ANINT (A) has the value AINT (A – 0.5).

Examples. ANINT (2.783) has the value 3.0. ANINT (–2.783) has the value –3.0.

13.14.11 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 $1 \leq \mathrm{DIM} \leq n$, 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 or MASK has rank one; otherwise, the result is an array 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 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) has a value equal to that of ANY (MASK). Otherwise, the value of element $(s_1, s_2, ..., s_{\mathrm{DIM}-1}, s_{\mathrm{DIM}+1}, ..., s_n)$ of ANY (MASK, DIM) is equal to ANY (MASK $(s_1, s_2, ..., s_{\mathrm{DIM}-1}, :, s_{\mathrm{DIM}+1}, ..., s_n)$).

Examples.

Case (i): The value of ANY ((/ .TRUE., .FALSE., .TRUE. /)) is true.

Case (ii): If B is the array $\begin{bmatrix}1 & 3 & 5\\ 2 & 4 & 6\end{bmatrix}$ and C is the array $\begin{bmatrix}0 & 3 & 5\\ 7 & 4 & 8\end{bmatrix}$ ANY (B .NE. C, DIM = 1) is [true, false, true] and ANY (B .NE. C, DIM = 2) is [true, true].

13.14.12 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).

13.14.13 ASSOCIATED (POINTER [, TARGET])

Description. Returns the association status of its pointer argument or indicates whether or not the pointer is associated with the target.

Class. Inquiry function.

Arguments.

POINTER	shall be a pointer and may be of any type. Its pointer association status shall not be undefined.
TARGET (optional)	shall be a pointer or target. It shall have the same type, type parameters, and rank as POINTER. If it is a pointer, its pointer association status shall not be undefined.

Result Characteristics. The result is of type default logical scalar.

Result Value.

Case (i): If TARGET is absent, the result is true if POINTER is currently associated with a target and false if it is not.

Case (ii): 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 (iii): 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 (iv): 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 (v): 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 points to 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.

13.14.14 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 $-\pi/2 \le \mathrm{ATAN}(\mathrm{X}) \le \pi/2$.

Example. ATAN (1.5574077) has the value 1.0 (approximately).

13.14.15 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 $-\pi < ATAN2(Y, X) \le \pi$ and is equal to a processor-dependent approximation to a value of arctan(Y/X) if $\mathrm{X} \neq 0$. If $\mathrm{Y} > 0$, the result is positive. If $\mathrm{Y} = 0$, the result is zero if $\mathrm{X} > 0$ and the result is π if $\mathrm{X} < 0$. If $\mathrm{Y} < 0$, the result is negative. If $\mathrm{X} = 0$, 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 $\begin{bmatrix} \frac{3\pi}{4} & \frac{\pi}{4} \\ \frac{-3\pi}{4} & -\frac{\pi}{4} \end{bmatrix}$.

13.14.16 BIT_SIZE (I)

Description. Returns the number of bits z defined by the model of 13.5.7.

Class. Inquiry function.

Argument. I shall be of type integer. It may be scalar or array valued.

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 13.5.7.

Example. BIT_SIZE (1) has the value 32 if z of the model is 32.

13.14.17 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. The result is of type 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 13.5.7.

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 $\begin{bmatrix} \text{true} & \text{false} \\ \text{false} & \text{false} \end{bmatrix}$.

13.14.18 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 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.

13.14.19 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 function ICHAR.

Class. Elemental function.

Arguments.

I	shall be of type integer with a value in the range $0 \le \text{I} \le n-1$, 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 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 $0 \leq I \leq n-1$ 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.

13.14.20 CMPLX (X [, Y, KIND])

Description. Convert to complex type.

Class. Elemental function.

Arguments.

X	shall be of type integer, real, or complex.
Y (optional)	shall be of type integer or real. 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 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 Y is absent and X is complex, it is as if Y were present with the value AIMAG (X). 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).

13.14.21 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).

13.14.22 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).

13.14.23 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).

13.14.24 COUNT (MASK [, DIM])

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 $1 \leq \mathrm{DIM} \leq n$, 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 default integer. It is scalar if DIM is absent or MASK has rank one; otherwise, the result is an array of rank $n-1$ and of shape $(d_1, d_2, ..., d_{\mathrm{DIM}-1}, d_{\mathrm{DIM}+1}, ..., d_n)$ 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 $(s_1, s_2, ..., s_{\mathrm{DIM}-1}, s_{\mathrm{DIM}+1}, ..., s_n)$ of COUNT (MASK, DIM) is equal to COUNT (MASK $(s_1, s_2, ..., s_{\mathrm{DIM}-1}, :, s_{\mathrm{DIM}+1}, ..., s_n)$).

Examples.

Case (i): The value of COUNT ((/ .TRUE., .FALSE., .TRUE. /)) is 2.

Case (i): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ and C is the array $\begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$, COUNT (B .NE. C, DIM = 1) is [2, 0, 1] and COUNT (B .NE. C, DIM = 2) is [1, 2].

13.14.25 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 13.8

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 computer or discover which parts of a calculation on a computer 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 may or may not include system overhead, and has no obvious connection to elapsed "wall clock" time.

13.14.26 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 $(d_1, d_2, ..., d_{\mathrm{DIM}-1}, d_{\mathrm{DIM}+1}, ..., d_n)$ 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 $1 \le \mathrm{DIM} \le n$, 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 $(s_1, s_2, ..., s_{\mathrm{DIM}-1}, :, s_{\mathrm{DIM}+1}, ..., s_n)$ of the result has a value equal to CSHIFT (ARRAY $(s_1, s_2, ..., s_{\mathrm{DIM}-1}, :, s_{\mathrm{DIM}+1}, ..., s_n)$, sh, 1), where sh is SHIFT or SHIFT $(s_1, s_2, ..., s_{\mathrm{DIM}-1}, s_{\mathrm{DIM}+1}, ..., s_n)$.

Examples.

Case (i): If V is the array [1, 2, 3, 4, 5, 6], the effect of shifting V 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 $\begin{bmatrix} 3 & 1 & 2 \\ 5 & 6 & 4 \\ 7 & 8 & 9 \end{bmatrix}$.

13.14.27 DATE_AND_TIME ([DATE, TIME, ZONE, VALUES])

Description. Returns data on 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, and shall be of length at least 8 in order to contain the complete value. It is an INTENT (OUT) argument. Its leftmost 8 characters are assigned a value of the form *CCYYMMDD*, where *CC* is the century, *YY* the year within the century, *MM* the month within the year, and *DD* the day within the month. If there is no date available, they are assigned blanks.

TIME (optional) shall be scalar and of type default character, and shall be of length at least 10 in order to contain the complete value. It is an INTENT (OUT) argument. Its leftmost 10 characters are assigned a value of the form *hhmmss.sss*, where *hh* is the hour of the day, *mm* is the minutes of the hour, and *ss.sss* is the seconds and milliseconds of the minute. If there is no clock available, they are assigned blanks.

ZONE (optional) shall be scalar and of type default character, and shall be of length at least 5 in order to contain the complete value. It is an INTENT (OUT) argument. Its leftmost 5 characters are assigned a value of the form ±*hhmm*, where *hh* and *mm* are the time difference with respect to Coordinated Universal Time (UTC) in hours and parts of an hour expressed in minutes, respectively. If there is no clock available, they are assigned blanks.

VALUES (optional) shall be of type default integer and of rank one. It is an INTENT (OUT) argument. Its size shall be at least 8. The values returned in VALUES are as follows:

- VALUES (1) the year (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 called in Geneva, Switzerland on 1985 April 12 at 15:27:35.5 would have assigned the value 19850412 to BIG_BEN (1), the value 152735.500 to BIG_BEN (2), and the value +0100 to BIG_BEN (3), and the following values to DATE_TIME: 1985, 4, 12, 60, 15, 27, 35, 500.

NOTE 13.9

UTC is defined by ISO 8601:1988 (and is also known as Greenwich Mean Time).

13.14.28 DBLE (A)

Description. Convert to double precision real type.

Class. Elemental function.

Argument. A shall be of type integer, real, or complex.

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.

13.14.29 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 scalar or array valued.

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 13.7.1 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 at the end of 13.7.1.

13.14.30 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.

13.14.31 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 array valued and of rank one.

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 array valued and of rank one. 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. 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. 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.

13.14.32 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.

13.14.33 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

$(d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n)$ 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 $(d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n)$. BOUNDARY may be omitted for the data types in the following table and, in this case, it is as if it were present with the scalar value shown.

Type of ARRAY	Value of BOUNDARY
Integer	0
Real	0.0
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 $1 \le DIM \le n$, 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 $(s_1, s_2, ..., s_{DIM-1}, s_{DIM} + sh, s_{DIM+1}, ..., s_n)$ where sh is SHIFT or SHIFT $(s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n)$ provided the inequality LBOUND (ARRAY, DIM) $\le s_{DIM} + sh \le$ UBOUND (ARRAY, DIM) holds and is otherwise BOUNDARY or BOUNDARY $(s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n)$.

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 $\begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix}$, then the value of

EOSHIFT (M, SHIFT = –1, BOUNDARY = '*', DIM = 2) is $\begin{bmatrix} * & A & B \\ * & D & E \\ * & G & H \end{bmatrix}$, and the value of EOSHIFT (M, SHIFT = (/ –1, 1, 0 /), BOUNDARY = (/ '*', '/', '?' /), DIM = 2) is $\begin{bmatrix} * & A & B \\ E & F & / \\ G & H & I \end{bmatrix}$.

13.14.34 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 scalar or array valued.

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 13.7.1 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 at the end of 13.7.1.

13.14.35 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).

13.14.36 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 (13.7.1) for the value of X, provided X is nonzero and e is within the range for default integers. EXPONENT (X) has the value zero if X is zero.

Examples. EXPONENT (1.0) has the value 1 and EXPONENT (4.1) has the value 3 for reals whose model is as at the end of 13.7.1.

13.14.37 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 KIND; otherwise, the kind type parameter is that of default integer type.

Result Value. The result has 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.

13.14.38 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 13.7.1 for the model representation of X. If X has the value zero, the result has the value zero.

Example. FRACTION (3.0) has the value 0.75 for reals whose model is as at the end of 13.7.1.

13.14.39 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 scalar or array valued.

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 13.7.1 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 at the end of 13.7.1.

13.14.40 IACHAR (C)

Description. Returns the position of a character in the ASCII collating sequence.

Class. Elemental function.

Argument. C shall be of type default character and of length one.

Result Characteristics. Default integer.

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 ($0 \leq \mathrm{IACHAR(C)} \leq 127$). 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) **.**LE**.** IACHAR (D) is true where C and D are any two characters representable by the processor.

Example. IACHAR ('X') has the value 88.

13.14.41 IAND (I, J)

Description. Performs a logical 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

I	J	IAND (I, J)
0	1	0
0	0	0

The model for the interpretation of an integer value as a sequence of bits is in 13.5.7.

Example. IAND (1, 3) has the value 1.

13.14.42 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 13.5.7.

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].

13.14.43 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 13.5.7.

Example. IBITS (14, 1, 3) has the value 7.

13.14.44 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 13.5.7.

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].

13.14.45 ICHAR (C)

Description. Returns the position of a character in the processor collating sequence associated with the kind type parameter of the character.

Class. Elemental function.

Argument. C shall be of type character and of length one. Its value shall be that of a character capable of representation in the processor.

Result Characteristics. Default integer.

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 $0 \leq \mathrm{ICHAR(C)} \leq n-1$, where n is the number of characters in the collating sequence. For any characters C and D capable of representation in the processor, C **.**LE**.** D is true if and only if ICHAR (C) **.**LE**.** ICHAR (D) is true and C **.**EQ**.** D is true if and only if ICHAR (C) **.**EQ**.** ICHAR (D) is true.

Example. ICHAR ('X') has the value 88 on a processor using the ASCII collating sequence for the default character type.

13.14.46 IEOR (I, J)

Description. Performs an 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 13.5.7.

Example. IEOR (1, 3) has the value 2.

13.14.47 INDEX (STRING, SUBSTRING [, BACK])

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.

Result Characteristics. Default integer.

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.

13.14.48 INT (A [, KIND])

Description. Convert to integer type.

Class. Elemental function.

Arguments.

A	shall be of type integer, real, or complex.
KIND (optional)	shall be a scalar integer initialization expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by 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) is the value obtained by applying the case (ii) rule to the real part of A.

Example. INT (–3.7) has the value –3.

13.14.49 IOR (I, J)

Description. Performs an 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 13.5.7.

Example. IOR (1, 3) has the value 3.

13.14.50 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 13.5.7.

Example. ISHFT (3, 1) has the result 6.

13.14.51 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.
SIZE (optional)	shall be of type integer. The value of SIZE shall be positive and shall not exceed BIT_SIZE (I). If SIZE is absent, it is as if it were present with the value of BIT_SIZE (I).

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 13.5.7.

Example. ISHFTC (3, 2, 3) has the value 5.

13.14.52 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 scalar or array valued.

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.

13.14.53 LBOUND (ARRAY [, DIM])

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 a pointer that is disassociated or an allocatable array that is not allocated.

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. The corresponding actual argument shall not be an optional dummy argument.

Result Characteristics. The result is of type default integer. It is scalar if DIM is present; otherwise, the result is an array of rank one and size n, where n is the rank of ARRAY.

Result Value.

Case (i): For an array section or for an array expression other than a whole array or array structure component, LBOUND (ARRAY, DIM) has the value 1. For a whole array or array structure component, LBOUND (ARRAY, DIM) has the value:

(a) equal to the lower bound for subscript DIM of ARRAY if dimension DIM of ARRAY does not have extent zero or if ARRAY is an assumed-size array of rank DIM, or

(b) 1 otherwise.

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.

13.14.54 LEN (STRING)

Description. Returns the length of a character entity.

Class. Inquiry function.

Argument. STRING shall be of type character. It may be scalar or array valued.

Result Characteristics. Default integer scalar.

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 array valued.

Example. If C is declared by the statement

```
CHARACTER (11) C (100)
```

LEN (C) has the value 11.

13.14.55 LEN_TRIM (STRING)

Description. Returns the length of the character argument without counting trailing blank characters.

Class. Elemental function.

Argument. STRING shall be of type character.

Result Characteristics. Default integer.

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.

13.14.56 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 13.10

The result is true if both STRING_A and STRING_B are of zero length.

Example. LGE ('ONE', 'TWO') has the value false.

13.14.57 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 13.11

The result is false if both STRING_A and STRING_B are of zero length.

Example. LGT ('ONE', 'TWO') has the value false.

13.14.58 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 13.12

The result is true if both STRING_A and STRING_B are of zero length.

Example. LLE ('ONE', 'TWO') has the value true.

13.14.59 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 13.13

The result is false if both STRING_A and STRING_B are of zero length.

Example. LLT ('ONE', 'TWO') has the value true.

13.14.60 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 $-\pi < \omega \leq \pi$. The imaginary part of the result is π only when the real part of the argument is less than zero and the imaginary part of the argument is zero.

Example. LOG (10.0) has the value 2.3025851 (approximately).

13.14.61 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).

13.14.62 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 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.

13.14.63 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 array valued and 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 array valued and 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 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 arguments according to 7.1.4.2. 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].

13.14.64 MAX (A1, A2 [, A3, ...])

Description. Maximum value.

Class. Elemental function.

Arguments. The arguments shall all have the same type which shall be integer or real and they shall all have the same kind type parameter.

Result Characteristics. Same as the arguments.

Result Value. The value of the result is that of the largest argument.

Example. MAX (–9.0, 7.0, 2.0) has the value 7.0.

13.14.65 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 scalar or array valued.

Result Characteristics. Default integer scalar.

Result Value. The result has the value $e_{\max}$, as defined in 13.7.1 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 at the end of 13.7.1.

13.14.66 MAXLOC (ARRAY, DIM [, MASK]) or MAXLOC (ARRAY [, MASK])

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 or real. It shall not be scalar.
DIM	shall be scalar and of type integer with a value in the range $1 \le \mathrm{DIM} \le n$, 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 type default integer. 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 *i*th subscript returned lies in the range 1 to e_i, where e_i is the extent of the *i*th dimension of ARRAY. If more than one element has the maximum value, the element whose subscripts are returned is the first such element, taken in array element order. If ARRAY has size zero, the value of the result is processor dependent.

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 *i*th subscript returned lies in the range 1 to e_i, where e_i is the extent of the *i*th dimension of ARRAY. If more than one such element has the maximum value, the element whose subscripts are returned is the first such element taken in array element order. If there are no such elements (that is, if ARRAY has size zero or every element of MASK has the value false), the value of the result is processor dependent.

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 $(s_1, s_2, ..., s_{\mathrm{DIM}-1}, :, s_{\mathrm{DIM}+1}, ..., s_n)$]).

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 **.**LT**.** 6) has the value [3, 2]. Note that this is true even if A has a declared lower bound other than 1.

Case (iii): The value of MAXLOC ((/ 5, -9, 3 /), DIM = 1) is 1. If B has the value $\begin{bmatrix} 1 & 3 & -9 \\ 2 & 2 & 6 \end{bmatrix}$, MAXLOC (B, DIM = 1) is [2, 1, 2] and MAXLOC (B, DIM = 2) is [2, 3]. Note that this is true even if B has a declared lower bound other than 1.

13.14.67 MAXVAL (ARRAY, DIM [, MASK]) or MAXVAL (ARRAY [, MASK])

Description. Maximum value of the elements of ARRAY along dimension DIM corresponding to the true elements of MASK.

Class. Transformational function.

Arguments.

ARRAY	shall be of type integer or real. It shall not be scalar.
DIM	shall be scalar and of type integer with a value in the range $1 \le \mathrm{DIM} \le n$, 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 or ARRAY has rank one; otherwise, the result is an array 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.

Result Value.

Case (i): The result of MAXVAL (ARRAY) has a value equal to the maximum value of all the elements of ARRAY or 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.

Case (ii): The result of MAXVAL (ARRAY, MASK = MASK) has a value equal to the maximum value of the elements of ARRAY corresponding to true elements of MASK or 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 there are no true elements.

Case (iii): If ARRAY has rank one, MAXVAL (ARRAY, DIM = DIM [,MASK = MASK]) has a value equal to that of MAXVAL (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

MAXVAL (ARRAY $(s_1, s_2, ..., s_{\mathrm{DIM}-1}, :, s_{\mathrm{DIM}+1}, ..., s_n)$
[, MASK = MASK $(s_1, s_2, ..., s_{\mathrm{DIM}-1}, :, s_{\mathrm{DIM}+1}, ..., s_n)$]).

Examples.

Case (i): The value of MAXVAL ((/ 1, 2, 3 /)) is 3.

Case (ii): MAXVAL (C, MASK = C **.**LT**.** 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].

13.14.68 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 $\begin{bmatrix}\text{T} & . & \text{T}\\ . & . & \text{T}\end{bmatrix}$, 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.

13.14.69 MIN (A1, A2 [, A3, ...])

Description. Minimum value.

Class. Elemental function.

Arguments. The arguments shall all be of the same type which shall be integer or real and they shall all have the same kind type parameter.

Result Characteristics. Same as the arguments.

Result Value. The value of the result is that of the smallest argument.

Example. MIN (–9.0, 7.0, 2.0) has the value –9.0.

13.14.70 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 scalar or array valued.

Result Characteristics. Default integer scalar.

Result Value. The result has the value $e_{\min}$, as defined in 13.7.1 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 at the end of 13.7.1.

13.14.71 MINLOC (ARRAY, DIM [, MASK]) or MINLOC (ARRAY [, MASK])

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 or real. It shall not be scalar.
DIM	shall be scalar and of type integer with a value in the range $1 < \text{DIM} \le n$, 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 type default integer. 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 *i*th subscript returned lies in the range 1 to e_i, where e_i is the extent of the *i*th dimension of ARRAY. If more than one element has the minimum value, the element whose subscripts are returned is the first such element, taken in array element order. If ARRAY has size zero, the value of the result is processor dependent.

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 *i*th subscript returned lies in the range 1 to e_i, where e_i is the extent of the *i*th dimension of ARRAY. If more than one such element has 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, the value of the result is processor dependent.

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 $(s_1, s_2, ..., s_{\mathrm{DIM}-1}, s_{\mathrm{DIM}+1}, ..., s_n)$ of the result is equal to

MINLOC (ARRAY $(s_1, s_2, ..., s_{\mathrm{DIM}-1}, :, s_{\mathrm{DIM}+1}, ..., s_n)$, DIM=1
[, MASK = MASK $(s_1, s_2, ..., s_{\mathrm{DIM}-1}, :, s_{\mathrm{DIM}+1}, ..., s_n)$]).

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 **.**GT**.** –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.

13.14.72 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 or real. It shall not be scalar.
DIM	shall be scalar and of type integer with a value in the range $1 \le \mathrm{DIM} \le n$, 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 or ARRAY has rank one; otherwise, the result is an array of rank

$n-1$ and of shape $(d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n)$ 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 or 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.

Case (ii): The result of MINVAL (ARRAY, MASK = MASK) has a value equal to the minimum value of the elements of ARRAY corresponding to true elements of MASK or 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 there are no true elements.

Case (iii): If ARRAY has rank one, MINVAL (ARRAY, DIM = DIM [, MASK = MASK]) has a value equal to that of MINVAL (ARRAY [, MASK = MASK]). Otherwise, the value of element $(s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n)$ of the result is equal to

MINVAL (ARRAY $(s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n)$
[, MASK= MASK $(s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n)$]).

Examples.

Case (i): The value of MINVAL ((/ 1, 2, 3 /)) is 1.

Case (ii): MINVAL (C, MASK = C **.**GT**.** 0.0) forms the minimum of the positive elements of C.

Case (iii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$, MINVAL (B, DIM = 1) is [1, 3, 5] and MINVAL (B, DIM = 2) is [1, 2].

13.14.73 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.

Result Characteristics. Same as A.

Result Value. If P ≠ 0, the value of the result is A–INT (A/P) * P. If P = 0, the result is processor dependent.

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.

13.14.74 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.

Result Characteristics. Same as A.

Result Value.

Case (i): A is of type integer. If $P \neq 0$, 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 $P < 0$. If P = 0, the result is processor dependent.

Case (ii): A is of type real. If $P \neq 0$, the value of the result is A – FLOOR (A / P) * P. If P = 0, the result is processor dependent.

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.

13.14.75 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 13.5.7.
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 the same variable as FROM. 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 13.5.7.
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.

13.14.76 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 at the end of 13.7.1.

13.14.77 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 KIND; otherwise, the kind type parameter is that of default integer type.

Result Value. If A > 0, NINT (A) has the value INT (A+0.5); if A ≤ 0, NINT (A) has the value INT (A–0.5).

Example. NINT (2.783) has the value 3.

13.14.78 NOT (I)

Description. Performs a logical 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 13.5.7.

Example. If I is represented by the string of bits 01010101, NOT (I) has the binary value 10101010.

13.14.79 NULL ([MOLD])

Description. Returns a disassociated pointer.

Class. Transformational function.

Argument. MOLD shall be a pointer and may be of any type. Its pointer association status may be undefined, disassociated, or associated. If its status is associated, the target need not be defined with a value.

Result Characteristics. The same as MOLD if MOLD is present; otherwise, determined by context (7.1.4.1).

Result. The result is a pointer with disassociated association status.

Example. REAL, POINTER, DIMENSION(:) :: VEC => NULL ( ) defines the initial association status of VEC to be disassociated.

13.14.80 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 **.**NE**.** 0) is [9, 7] and the result of PACK (M, M **.**NE**.** 0, VECTOR = (/ 2, 4, 6, 8, 10, 12 /)) is [9, 7, 6, 8, 10, 12].

13.14.81 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 scalar or array valued.

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 13.7.1 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 at the end of 13.7.1.

13.14.82 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 scalar or array valued. 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 (12.4.1.5) and otherwise has the value false.

13.14.83 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 $1 \le \mathrm{DIM} \le n$, 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 or ARRAY has rank one; otherwise, the result is an array 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.

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 $(s_1, s_2, ..., s_{\mathrm{DIM}-1}, s_{\mathrm{DIM}+1}, ..., s_n)$ of PRODUCT (ARRAY, DIM = DIM [,MASK = MASK]) is equal to

PRODUCT (ARRAY $(s_1, s_2, ..., s_{\mathrm{DIM}-1}, :, s_{\mathrm{DIM}+1}, ..., s_n)$
[, MASK = MASK $(s_1, s_2, ..., s_{\mathrm{DIM}-1}, :, s_{\mathrm{DIM}+1}, ..., s_n)$]).

Examples.

Case (i): The value of PRODUCT ((/ 1, 2, 3 /)) is 6.

Case (ii): PRODUCT (C, MASK = C **.**GT**.** 0.0) forms the product of the positive elements of C.

Case (iii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$, PRODUCT (B, DIM = 1) is [2, 12, 30] and PRODUCT (B, DIM = 2) is [15, 48].

13.14.84 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 scalar or array valued.

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 13.7.1 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 at the end of 13.7.1.

13.14.85 RANDOM_NUMBER (HARVEST)

Description. Returns one pseudorandom number or an array of pseudorandom numbers from the uniform distribution over the range $0 \le x < 1$.

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 $0 \le x < 1$.

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
```

13.14.86 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 $\ge N$. 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 $\ge N$ 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
```

13.14.87 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 scalar or array valued.

Result Characteristics. Default integer scalar.

Result Value.

Case (i): For an integer argument, the result has the value INT (LOG10 (*huge*)), where *huge* is the largest positive integer of the model representing integer numbers with same kind type parameter as X (13.7.1).

Case (ii): For a real or complex argument, the result has the value INT (MIN (LOG10 (*huge*), –LOG10 (*tiny*))), where *huge* and *tiny* are the largest and smallest positive numbers of the model representing real numbers with the same value for the kind type parameter as X (13.7.1).

Examples. RANGE (X) has the value 38 for real X whose model is as at the end of 13.7.1, since in this case *huge* = $(1 - 2^{-24}) \times 2^{127}$ and *tiny* = 2^{-127}.

13.14.88 REAL (A [, KIND])

Description. Convert to real type.

Class. Elemental function.

Arguments.

A	shall be of type integer, real, or complex.
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 KIND. If A is of type integer or real and KIND is not present, the kind type parameter is the processor-dependent kind type parameter for the default real type.

Case (ii): If A is of type complex and KIND is present, the kind type parameter is that specified by KIND. If A is of type complex and KIND is not present, the kind type parameter is the kind type parameter of A.

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.

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.

13.14.89 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.

13.14.90 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 be array valued. 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 be array valued.
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 $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 0 & 0 \end{bmatrix}$.

13.14.91 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 13.7.1 for the model representation of X.

Example. RRSPACING (–3.0) has the value 0.75×2^{24} for reals whose model is as at the end of 13.7.1.

13.14.92 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 13.7.1 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 at the end of 13.7.1.

13.14.93 SCAN (STRING, SET [, BACK])

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.

Result Characteristics. Default integer.

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.

13.14.94 SELECTED_INT_KIND (R)

Description. Returns a value of the kind type parameter of an integer data type that represents all integer values n with $-10^R < n < 10^R$.

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 data type that represents all values n in the range of values n with $-10^R < n < 10^R$, 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 criteria, 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. SELECTED_INT_KIND (6) has the value KIND (0) on a machine that supports a default integer representation method with $r = 2$ and $q = 31$.

13.14.95 SELECTED_REAL_KIND ([P, R])

Description. Returns a value of the kind type parameter of a real data 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. The result has a value equal to a value of the kind type parameter of a real data 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 data type with a precision greater than or equal to P, –2 if the processor does not support a real type with an exponent range greater than or equal to R, and –3 if neither is supported. 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 $e_{\max} = 63$.

13.14.96 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 $X \times b^{I-e}$, where b and e are as defined in 13.7.1 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 at the end of 13.7.1.

13.14.97 SHAPE (SOURCE)

Description. Returns the shape of an array or a scalar.

Class. Inquiry function.

Argument. SOURCE may be of any type. It may be array valued or scalar. It shall not be a pointer that is disassociated or an allocatable array that is not allocated. It shall not be an assumed-size array.

Result Characteristics. The result is a default integer 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.

13.14.98 SIGN (A, B)

Description. Absolute value of A times 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

(a) If the processor cannot distinguish between positive and negative real zero, the value of the result is $|A|$.

(b) If B is positive real zero, the value of the result is $|A|$.

(c) If B is negative real zero, the value of the result is $-|A|$.

Example. SIGN (–3.0, 2.0) has the value 3.0.

13.14.99 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).

13.14.100 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).

13.14.101 SIZE (ARRAY [, DIM])

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 a pointer that is disassociated or an allocatable array that is not allocated. 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 $1 \leq \mathrm{DIM} \leq n$, where n is the rank of ARRAY.

Result Characteristics. Default integer scalar.

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.

13.14.102 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 is not zero, the result has the value b^{e-p}, where b, e, and p are as defined in 13.7.1 for the model representation of X, provided this result is within range. Otherwise, the result is the same as that of TINY (X).

Example. SPACING (3.0) has the value 2^{-22} for reals whose model is as at the end of 13.7.1.

13.14.103 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 scalar or array valued. The rank of SOURCE shall be less than 7.

DIM shall be scalar and of type integer with value in the range $1 \le \mathrm{DIM} \le n+1$, 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 array valued with shape $(d_1, d_2, ..., d_n)$, the shape of the result is $(d_1, d_2, ..., d_{\mathrm{DIM}-1}, \mathrm{MAX}(\mathrm{NCOPIES}, 0), d_{\mathrm{DIM}}, ..., d_n)$.

Result Value.

Case (i): If SOURCE is scalar, each element of the result has a value equal to SOURCE.

Case (ii): If SOURCE is array valued, the element of the result with subscripts $(r_1, r_2, ..., r_{n+1})$ has the value SOURCE $(r_1, r_2, ..., r_{\mathrm{DIM}-1}, r_{\mathrm{DIM}+1}, ..., r_{n+1})$.

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.

13.14.104 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 is greater than or equal to zero.

Example. SQRT (4.0) has the value 2.0 (approximately).

13.14.105 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 $1 \le \mathrm{DIM} \le n$, 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 or ARRAY has rank one; otherwise, the result is an array 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.

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 $(s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n)$ of SUM (ARRAY, DIM = DIM [, MASK = MASK]) is equal to

SUM (ARRAY $(s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n)$
[, MASK= MASK $(s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n)$]).

Examples.

Case (i): The value of SUM ((/ 1, 2, 3 /)) is 6.

Case (ii): SUM (C, MASK= C **.**GT**.** 0.0) forms the arithmetic sum of the positive elements of C.

Case (iii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$, SUM (B, DIM = 1) is [3, 7, 11] and SUM (B, DIM = 2) is [9, 12].

13.14.106 SYSTEM_CLOCK ([COUNT, COUNT_RATE, COUNT_MAX])

Description. Returns integer data from a real-time clock.

Class. Subroutine.

Arguments.

COUNT (optional) shall be scalar and of type default 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 (0) if there is no clock. The processor-dependent value is incremented by one for each clock count until the value COUNT_MAX is reached and is reset to zero at the next count. It lies in the range 0 to COUNT_MAX if there is a clock.

COUNT_RATE (optional)
shall be scalar and of type default integer. It is an INTENT (OUT) argument. It is assigned a processor-dependent approximation to the number of processor clock counts per second, or zero if there is no clock.

COUNT_MAX (optional)
shall be scalar and of type default integer. It is an INTENT (OUT) argument. It is assigned the maximum value that COUNT can have, or zero if there is no clock.

Example. If the processor clock is a 24-hour clock that registers time in 1-second intervals, at 11:30 A.M. the reference

```
CALL SYSTEM_CLOCK (COUNT = C, COUNT_RATE = R, COUNT_MAX = M)
```

defines C = 11 × 3600 + 30 × 60 = 41400, R = 1, and M = 24 × 3600 – 1 = 86399.

13.14.107 TAN (X)

Description. 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 tan(X), with X regarded as a value in radians.

Example. TAN (1.0) has the value 1.5574077 (approximately).

13.14.108 TANH (X)

Description. Hyperbolic 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 tanh(X).

Example. TANH (1.0) has the value 0.76159416 (approximately).

13.14.109 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 scalar or array valued.

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 13.7.1 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 at the end of 13.7.1.

13.14.110 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 and may be scalar or array valued.
MOLD	may be of any type and may be scalar or array valued.
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 array valued and SIZE is absent, the result is array valued 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 array valued 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 undefined. 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 undefined.

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).

13.14.111 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 $\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$.

13.14.112 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'.

13.14.113 UBOUND (ARRAY [, DIM])

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 a pointer that is disassociated or an allocatable array that is not allocated. 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 $1 \le \mathrm{DIM} \le n$, where n is the rank of ARRAY. The corresponding actual argument shall not be an optional dummy argument.

Result Characteristics. The result is of type default integer. It is scalar if DIM is present; otherwise, the result is an array of rank one and size n, where n is the rank of ARRAY.

Result Value.

Case (i): 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 component 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.

13.14.114 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 array valued and 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. Specific values may be "scattered" to specific 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 UNPACK (V, MASK = Q, FIELD = 0) has the value $\begin{bmatrix} 0 & 2 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 3 \end{bmatrix}$.

13.14.115 VERIFY (STRING, SET [, BACK])

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.

Result Characteristics. Default integer.

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.

Section 14: Scope, association, and definition

Entities are identified by lexical tokens within a **scope** that is a program, a scoping unit, a construct, a single statement, or part of a statement. If the scope is a program, the entity is called a **global entity**. If the scope is a scoping unit (2.2), the entity is called a **local entity**. If the scope is a construct, the entity is called a **construct entity**. If the scope is a statement or part of a statement, the entity is called a **statement entity**.

An entity may be identified by

(1) A name (14.1),
(2) A label (14.2),
(3) An external input/output unit number (14.3),
(4) An operator symbol (14.4), or
(5) An assignment symbol (14.5).

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.

14.1 Scope of names

Named entities are global, local, construct, or statement entities.

14.1.1 Global entities

Program units, common blocks, and external procedures are global entities of a program. A name that identifies a global entity shall not be used to identify any other global entity in the same program.

NOTE 14.1

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, 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.

14.1.2 Local entities

Within a scoping unit, entities in the following classes:

(1) Named variables that are not statement or construct entities (14.1.3), named constants, named constructs, statement functions, internal procedures, module procedures, dummy procedures, intrinsic procedures, generic identifiers, derived types, and namelist group names,
(2) Type components, in a separate class for each type, and
(3) Argument keywords, in a separate class for each procedure with an explicit interface

are local entities of that scoping unit.

Except for a common block name (14.1.2.1), an external procedure name that is also a generic name (12.3.2.1), or an external function name within its defining subprogram (14.1.2.2), a name that identifies a global entity in a scoping unit shall not be used to identify a local entity of class (1) in that scoping unit.

Within a scoping unit, a name that identifies a local entity of one class shall not be used to identify another local entity of the same class, except in the case of generic names (12.3.2.1). A name that identifies a local entity of one class may be used to identify a local entity of another class.

NOTE 14.2

An intrinsic procedure is inaccessible in a scoping unit containing another local entity of the same class and having the same name. 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.

The name of a local entity identifies that entity in a scoping unit and may be used to identify any local or global 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 the name of 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 the name of 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 the name of a common block (the latter is possible only if the *entry-stmt* is in a module subprogram).

14.1.2.1 Common blocks

A common block name in a scoping unit also may be the name of any local entity other than a named constant, intrinsic procedure, or a local variable that is also an external function in a function subprogram. If a name is used for both a common block and a local entity, the appearance of that name in any context other than as a common block name in a COMMON or SAVE statement identifies only the local entity.

NOTE 14.3

An intrinsic procedure name may be a common block name in a scoping unit that does not reference the intrinsic procedure.

14.1.2.2 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.

14.1.2.3 Unambiguous generic procedure references

This subsection contains the rules that shall be satisfied by every pair of specific procedures that have the same generic name, have the same generic operator, or both define assignment. They ensure that a generic reference is unambiguous. When an intrinsic operator or assignment is extended, the rules apply as if the intrinsic consisted of a collection of specific procedures, one for each allowed combination of type, kind type parameter, and rank for each operand. When 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. If two or more generic interfaces that are accessible in a scoping unit have the same local name, the same operator, or are both assignments, they are interpreted as a single generic interface.

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 has a different type, different kind type parameter, or different rank.

Within a scoping unit, two procedures that have the same generic name shall both be subroutines or both be functions, and

(1) one of them shall have more nonoptional dummy arguments of a particular data type, kind type parameter, and rank than the other has dummy arguments (including optional dummy arguments) of that data type, kind type parameter, and rank; or

(2) at least one of them shall have both

  (a) A nonoptional dummy argument that corresponds by position in the argument list to a dummy argument not present in the other, present with a different type, present with a different kind type parameter, or present with a different rank; and

  (b) A nonoptional dummy argument that corresponds by argument keyword to a dummy argument not present in the other, present with a different type, present with a different kind type parameter, or present with a different rank.

  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 keyword.

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 14.4

The procedures with interface bodies given by the interface block

```
INTERFACE A
   SUBROUTINE AR (X)
      REAL X
   END SUBROUTINE AR

   SUBROUTINE AI (J)
      INTEGER J
   END SUBROUTINE AI
END INTERFACE A
```

NOTE 14.4 *(Continued)*

satisfy rules (2)(a) and (2)(b). However, if J were declared REAL, rule (2)(a) would not be satisfied while rule (2)(b) remains satisfied; in this case, the reference to A in the statement

```
CALL A (0.0)
```

would be ambiguous.

NOTE 14.5

If a reference to the intrinsic function NULL appears as an actual argument in a reference to a generic procedure, the argument MOLD may be required to resolve the reference (7.1.4.1).

14.1.2.4 Resolving 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 an interface body with that name;

  (b) if that scoping unit contains a module subprogram, internal subprogram, or statement function that defines a procedure with that name;

  (c) if that scoping unit contains an INTRINSIC attribute specification for that name and if it is the name of a specific intrinsic procedure;

  (d) if that scoping unit contains an EXTERNAL attribute specification for that name;

  (e) 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

  (f) 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 is not established in a scoping unit if it is neither established to be generic nor established to be specific.

14.1.2.4.1 Resolving procedure references to names established to be generic

(1) If the reference is consistent with 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 that interface block that provides that interface. The rules in 14.1.2.3 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 that interface block that provides that interface. The rules in 14.1.2.3 ensure that there can be at most one such specific interface.

NOTE 14.6

These rules allow specific instances of a generic function to be used for specific 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 14.7

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.

14.1.2.4.2 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 14.8

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.

14.1.2.4.3 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.

14.1.2.5 Components

A component name has the same scope as the type of which it is a component. Outside the type definition, it may appear only within a designator of a component of a structure of that type. If the type is accessible in another scoping unit by use association or host association (14.6.1.2, 14.6.1.3) and the definition of the type does not contain the PRIVATE statement (4.4.1), the component name is accessible for names of components of structures of that type in that scoping unit.

14.1.2.6 Argument keywords

A dummy argument name in an internal procedure, module procedure, or a procedure interface block has a scope as an argument keyword of the scoping unit of the host of the procedure or interface block. As an argument keyword, it may appear only in a procedure reference for the procedure of which it is a dummy argument. If the procedure or procedure interface block is accessible in another scoping unit by use association or host association (14.6.1.2, 14.6.1.3), 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 making reference to the procedure. As an argument keyword, it may appear only in a procedure reference for the procedure of which it is a dummy argument.

14.1.3 Statement and construct entities

The name of a variable that appears as the DO variable of an implied-DO in a DATA statement or an array constructor has a scope of the implied-DO list. 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 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 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 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.

Except for a common block name or a scalar variable name, a name that identifies a global entity or local entity of class 1 (14.1.2) accessible 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 the name of a global or local entity 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 that of the global or local entity.

Except for a common block name or a scalar variable name, a name that identifies a global entity or a local entity of class 1 (14.1.2) accessible in the scoping unit of a FORALL statement or FORALL construct shall not be the same as the *index-name*. Within the scope of a FORALL construct, a nested FORALL statement or FORALL construct shall not have the same *index-name*.

If the name of a global or local entity 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 that of the global or local entity.

14.2 Scope of labels

A label is a local entity. No two statements in the same scoping unit may have the same label.

14.3 Scope of external input/output units

An external input/output unit is a global entity.

14.4 Scope of operators

The intrinsic operators are global entities. A defined operator that is not an extended intrinsic operator is a local entity. Within a scoping unit an operator may identify additional operations as specified by the rules for generic operators (12.3.2.1).

14.5 Scope of the assignment symbol

The assignment symbol is a global entity. Within a scoping unit the assignment symbol may identify additional assignment operations or replace the intrinsic derived type assignment operation as specified by the rules for generic assignment (12.3.2.1).

14.6 Association

Two entities may become associated by name association, pointer association, or storage association.

14.6.1 Name association

There are three forms of **name association** : argument association, use association, and host association. Argument, use, and host association provide mechanisms by which entities known in one scoping unit may be accessed in another scoping unit.

14.6.1.1 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 (12.4.1.4).

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 14.9

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.

14.6.1.2 Use association

Use association is the association of names in different scoping units specified by a USE statement. The rules for use association are given in 11.3.2. They allow for the renaming of the entities being accessed. Use association allows access in one scoping unit to entities defined in another scoping unit and remains in effect throughout the execution of the program.

14.6.1.3 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**. The accessed entities are known by the same name and have the same attributes as in the host and are named data objects, derived types, interface blocks, 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. A name that is declared to be an external procedure name by an *external-stmt* or an *interface-body*, or that appears as a *module-name* in a *use-stmt* is a global name and 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*, or in a *target-stmt*;

(3) A *named-constant* in a *named-constant-def* in a *parameter-stmt*;

(4) An *array-name* in an *allocatable-stmt* or in a *dimension-stmt*;

(5) A *variable-name* in a *common-block-object* in a *common-stmt*;

(6) The name of a variable that is wholly or partially initialized in a *data-stmt*;

(7) The name of an object that is wholly or partially equivalenced in an *equivalence-stmt*;

(8) A *dummy-arg-name* in a *function-stmt*, in a *subroutine-stmt*, in an *entry-stmt*, or in a *stmt-function-stmt*;

(9) A *result-name* in a *function-stmt* or in an *entry-stmt*;

(10) An *intrinsic-procedure-name* in an *intrinsic-stmt*;

(11) A *namelist-group-name* in a *namelist-stmt*;

(12) A *generic-name* in a *generic-spec* in an *interface-stmt*; or

(13) The name of a named construct

is the name of a local entity 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 contains a subprogram or a derived type definition, the name of the subprogram or derived type is the name of a local entity and any entity of the host that has this as its nongeneric name is inaccessible by that name. Entities that are local (14.1.2) to a subprogram are not accessible to its host.

If a host entity is inaccessible only because a local entity with the same name is wholly or partially initialized in a DATA statement, the local entity 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 14.10

An interface body does not access the named entities by host association, but it may access entities by use association (11.3.2).

If a procedure gains access to a pointer by host association, the association of the pointer with a target that is current at the time the procedure is invoked remains current within the procedure. This pointer association may be changed within the procedure. When execution of the procedure completes, the pointer association that was current remains current, except where the completion causes the target to become undefined (item (3) of 14.7.6). In these cases, the completion of the procedure causes the pointer association status of the host associated pointer to become undefined.

NOTE 14.11

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,
                       ! since 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 14.11 *(Continued)*

```
   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 14.12

For more examples of host association, see section C.10.1.

14.6.2 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.

NOTE 14.13

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 be in an undefined state. 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.

14.6.2.1 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.

14.6.2.1.1 Events that cause pointers to become associated

A pointer becomes associated as follows:

(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 (7.5.2) that is associated or is specified with the TARGET attribute and, if allocatable, is currently allocated.

14.6.2.1.2 Events that cause pointers to become disassociated

A pointer becomes disassociated as follows:

(1) The pointer is nullified (6.3.2),

(2) The pointer is deallocated (6.3.3), or

(3) The pointer is pointer-assigned to a disassociated pointer (7.5.2).

(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 function with this object as its result is invoked,

(b) a procedure with this object as an INTENT (OUT) dummy argument is invoked,

(c) a procedure with this object as an automatic data object is invoked,

(d) a procedure with this object as a local object that is not accessed by use or host association is invoked, or

(e) this object is allocated.

14.6.2.1.3 Events that cause the association status of pointers to become undefined

The following events cause the association status of a pointer to become undefined:

(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) Execution of a RETURN or END statement that causes the pointer's target to become undefined (item (3) of 14.7.6), or

(4) Execution of a RETURN or END statement in a subprogram where the pointer was either declared or, with the exceptions described in 6.3.3.2, accessed.

14.6.2.2 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 (14.7).

14.6.2.3 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 are determined by its target.

14.6.3 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.

14.6.3.1 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, or an unspecified storage unit.

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 one occupies one **character storage unit**;

(4) A nonpointer scalar object of type default character and character length *len* occupies *len* contiguous character storage units;

(5) A nonpointer scalar object of type nondefault integer, real other than default or double precision, nondefault logical, nondefault complex, nondefault character of any length, or nonsequence type occupies a single **unspecified storage unit** that is different for each case;

(6) A nonpointer array of intrinsic type or sequence derived type occupies a sequence of contiguous storage sequences, one for each array element, in array element order (6.2.2.2);

(7) A nonpointer scalar object of sequence type occupies a sequence of storage sequences corresponding to the sequence of its ultimate components; and

(8) A pointer occupies a single unspecified storage unit that is different from that of any nonpointer 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).

14.6.3.2 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 14.14

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
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.

14.6.3.3 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 named common block is permitted to contain a sequence of differing storage units provided each scoping unit that accesses the common block specifies an identical sequence of storage units. The same rule applies to blank common blocks. If the sizes of the two blank common blocks differ, the sequence of storage units of the shorter block shall be identical to the initial sequence of the storage units of the longer block.

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 14.15

In the example:

```
REAL A (4), B
COMPLEX C (2)
DOUBLE PRECISION D
EQUIVALENCE (C (2), A (2), B), (A, D)
```

the third storage unit of C, the second storage unit of A, the storage unit of B, and the second storage unit of D are specified as the same. The storage sequences may be illustrated as:

```
Storage unit       1     2     3     4     5
                ----C(1)----|---C(2)----
                      A(1)  A(2)  A(3)  A(4)
                             --B--
                    ------D------
```

A (2) and B are totally associated. The following are partially associated: A (1) and C (1), A (2) and C (2), A (3) and C (2), B and C (2), A (1) and D, A (2) and D, B and D, C (1) and D, and C (2) and D. 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 14.16

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 (4.4.1). Default initialization may be specified for a storage unit that is storage associated provided the objects or subobjects supplying the default initialization are of the same type and type parameters, and supply the same value for the storage unit.

14.7 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.

14.7.1 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 object is defined if and only if all of its subobjects are defined.

(2) If an object is undefined, at least one (but not necessarily all) of its subobjects are undefined.

14.7.2 Variables that are always defined

Zero-sized arrays and zero-length strings are always defined.

14.7.3 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 direct components of variables of a type in which default initialization is specified for those components, provided that the variables are not accessed by use or host association, do not have the ALLOCATABLE attribute or POINTER attribute, and either have the SAVE attribute or are declared in a main program, MODULE, or BLOCK DATA scoping unit, and

(4) Variables that are always defined.

14.7.4 Variables that are initially undefined

All other variables are initially undefined.

14.7.5 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 (7.5.3).

(3) As execution of an input statement proceeds, each variable that is assigned a value from the input file becomes defined at the time that data is transferred to it. (See (4) in 14.7.6.) Execution of a WRITE statement whose unit specifier identifies an internal file causes each record that is written to become defined.

(4) Execution of a DO statement causes the DO variable, if any, to become defined.

(5) Beginning of execution of the action specified by an implied-DO list in an input/output statement causes the implied-DO variable to become defined.

(6) A reference to a procedure causes the entire dummy argument data object to become defined if the entire corresponding actual argument is defined with a value that is not a statement label.

A reference to a procedure causes a subobject of a dummy argument to become defined if the corresponding subobject of the corresponding actual argument is defined.

(7) Execution of an input/output statement containing an IOSTAT= specifier causes the specified integer variable to become defined.

(8) Execution of a READ statement containing a SIZE= specifier causes the specified integer variable to become defined.

(9) 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.

(10) When a character storage unit becomes defined, all associated character storage units become defined.

When a numeric storage unit becomes defined, all associated numeric storage units of the same type become defined. When an entity of double precision real type becomes defined, all totally associated entities of double precision real type become defined.

When an unspecified storage unit becomes defined, all associated unspecified storage units become defined.

(11) When a default complex entity becomes defined, all partially associated default real entities become defined.

(12) When both parts of a default complex entity become defined as a result of partially associated default real or default complex entities becoming defined, the default complex entity becomes defined.

(13) When all components of a numeric sequence structure or character sequence structure become defined as a result of partially associated objects becoming defined, the structure becomes defined.

(14) Execution of an ALLOCATE or DEALLOCATE statement with a STAT= specifier causes the variable specified by the STAT= specifier to become defined.

(15) Allocation of a zero-sized array causes the array to become defined.

(16) Allocation of an object of a derived type, in which default initialization is specified for any nonpointer direct component, causes that component to become defined.

(17) Invocation of a procedure causes any automatic object of zero size in that procedure to become defined.

(18) Execution of a pointer assignment statement that associates a pointer with a target that is defined causes the pointer to become defined.

(19) Invocation of a procedure that contains a nonsaved local object that is not a dummy argument, is not accessed by use or host association, has neither the ALLOCATABLE nor POINTER attribute, and is of a derived type in which default initialization is specified for any direct components, causes those components of the object to become defined.

(20) Invocation of a procedure that has an INTENT (OUT) dummy argument of a derived type that specifies default initialization for a nonpointer direct component, causes that component of the dummy argument to become defined.

(21) Invocation of a nonpointer function of a derived type, in which default initialization is specified for a nonpointer direct component, causes that component of the function result to become defined.

(22) In a FORALL construct, the *index-name* becomes defined when the *index-name* value set is evaluated.

14.7.6 Events that cause variables to become undefined

Variables become undefined as follows:

(1) When a variable of a given type becomes defined, all associated variables of different type become undefined. However, when a variable of type default real is partially associated with a variable of type default complex, the complex variable does not become undefined when the real variable becomes defined and the real variable does not become undefined when the complex variable becomes defined. When 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 an argument of the function or a variable in a module or in a common block 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 argument or variable becomes undefined when the expression is evaluated.

(3) The execution of a RETURN statement or an END statement within a subprogram causes all variables local to its scoping unit or local to the current instance of its scoping unit for a recursive invocation to become undefined except for the following:

(a) Variables with the SAVE attribute.

(b) Variables in blank common.

(c) Variables in a named common block that appears in the subprogram and appears in at least one other scoping unit that is making either a direct or indirect reference to the subprogram.

(d) Variables accessed from the host scoping unit.

(e) Variables accessed from a module that also is referenced directly or indirectly by at least one other scoping unit that is making either a direct or indirect reference to the subprogram.

(f) Variables in a named common block that are initially defined (14.7.3) and that have not been subsequently defined or redefined.

(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 implied-DOs, all of the implied-DO variables in the statement become undefined (9.4.3).

(6) Execution of a defined assignment statement may leave all or part of the variable that precedes the equals undefined.

(7) 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.

(8) Execution of an INQUIRE statement may cause the NAME=, RECL=, and NEXTREC= variables to become undefined (9.6).

(9) 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.

(10) When an allocatable array is deallocated, it becomes undefined.

(11) Successful execution of an ALLOCATE statement for a non-zero-sized object for which default initialization has not been specified causes the object to become undefined.

(12) Execution of an INQUIRE statement causes all inquiry specifier variables to become undefined if an error condition exists, except for the variable in the IOSTAT= specifier, if any.

(13) When a procedure is invoked

(a) An optional dummy argument that is not associated with an actual argument is undefined;

(b) A dummy argument with INTENT (OUT) is undefined except for any nonpointer direct components of the argument for which default initialization is specified;

(c) An actual argument associated with a dummy argument with INTENT (OUT) becomes undefined;

(d) A subobject of a dummy argument that does not have INTENT (OUT) is undefined if the corresponding subobject of the actual argument is undefined; and

(e) The result variable of a function is undefined except for those nonpointer direct components of the result for which default initialization is specified.

(14) When the association status of a pointer becomes undefined or disassociated (6.3), the pointer becomes undefined.

(15) When the execution of a FORALL construct has completed, the *index-name* becomes undefined.

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. Where the definition uses a term that is itself defined in this glossary, the first occurrence of the term in that definition is printed in italics.

action statement (2.1) : A single *statement* specifying or controlling a computational action (R216).

actual argument (12.4.1) : An *expression*, a *variable*, a *procedure*, or an alternate return specifier that is specified in a procedure *reference*.

allocatable array (5.1.2.4.3) : A named *array* having the ALLOCATABLE *attribute*. Only when it has space allocated for it does it have a *shape* and may it be *referenced* or *defined*.

argument (12) : An *actual argument* or a *dummy argument*.

argument association (14.6.1.1) : The relationship between an *actual argument* and a *dummy argument* during the execution of a *procedure reference*.

argument keyword (2.5.2) : A *dummy argument name*. It may be used in a *procedure reference* followed by the equals symbol (R1212) provided the procedure has an *explicit interface*.

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 (5.1.2.4.3) : 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*.

array-valued : Having the property of being an *array.*

assignment statement (7.5.1.1) : A *statement* of the form "*variable = expression*".

association (14.6) : Name association, pointer association, or storage association.

assumed-shape array (5.1.2.4.2) : A nonpointer dummy array that takes its shape from the associated actual argument.

assumed-size array (5.1.2.4.4) : 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 nonconstant character length or array bound.

belong (8.1.4.4.3, 8.1.4.4.4) : 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.

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 (11.4) : A *program unit* that provides initial values for *data objects* in *named common blocks*.

bounds (5.1.2.4.1) : 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.

characteristics (12.2) :

(1) Of a *procedure*, its classification as a *function* or *subroutine*, 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*, or has the OPTIONAL *attribute*.

(3) Of a *data object*, its *type*, *type parameters*, *shape*, the exact dependence of an array bound or the character length on other entities, *intent*, whether it is optional, whether it is a *pointer* or a *target*, and whether the shape, *size,* or *character length* is assumed.

(4) Of a *dummy procedure*, 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, whether it is a pointer, rank if it is a pointer, shape if it is not a pointer, the exact dependence of an array bound or the character length 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 (4.3.2.1) : A sequence of *characters* numbered from left to right 1, 2, 3, ...

character storage unit (14.6.3.1) : 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 (4.3.2.1.1) : 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*.

component (4.4) : A constituent of a *derived type*.

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 (9.3.2) :

(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*.

constant expression (7.1.6.1) : An *expression* satisfying rules that ensure that its value does not vary during *program* execution.

construct (7.5.3, 7.5.4, 8.1) : A sequence of *statements* starting with a SELECT CASE, DO, IF, FORALL, or WHERE statement and ending with the corresponding terminal statement.

construct entity (14) : An *entity* defined by a *lexical token* whose *scope* is a *construct*.

control mask (7.5.3) : 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 *data 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 (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. For an *intrinsic* type, the set of data values depends on the values of the *type parameters*.

datum : A single quantity that may have any of the set of values specified for its *data type*.

default initialization (4.4) : 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.

definable (2.5.4) : A *variable* is **definable** if its value may be changed by the appearance of its *name* or *designator* on the left of an *assignment statement*. An *allocatable array* 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 (2.5.4) : For a *data object*, the property of having or being given a valid value.

defined assignment statement (7.5.1.3) : An *assignment statement* that is not an *intrinsic* assignment statement and is defined by a *subroutine* and an *interface block* that specifies ASSIGNMENT (=).

defined operation (7.1.3) : An *operation* that is not an *intrinsic* operation and is defined by a *function* that is associated with a *generic identifier*.

deleted feature (1.7) : A feature in a previous Fortran standard that is considered to have been redundant and largely unused. See section B.1 for a list of features which were 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, 4.4) : A *type* whose *data* have *components*, each of which is either of *intrinsic* type or of another derived type.

designator : See *subobject designator*.

direct component (4.4) : The direct components of a *derived type* are

(1) The *components* of that *type* and

(2) For any nonpointer component that is of derived type, the direct components of that derived type.

disassociated (2.4.6) : A *pointer* is **disassociated** following execution of a DEALLOCATE or NULLIFY *statement*, or following *pointer association* with a disassociated pointer.

dummy argument (12.5.2.2, 12.5.2.3, 12.5.2.5, 12.5.4) : An entity whose *name* appears in the parenthesized list following the *procedure* name in a FUNCTION *statement*, a SUBROUTINE *statement*, an ENTRY statement, or a *statement function* statement.

dummy array : A *dummy argument* that is an *array*.

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*.

elemental (2.4.5, 7.5.1.3, 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*, a *procedure*, an *operator*, an *interface block*, a *common block*, an *external unit*, a *statement function*, a *type*, a *data entity*, a *statement label*, a *construct*, or a namelist group.

executable construct (2.1) : A CASE, DO, FORALL, IF, or WHERE *construct* or an *action statement* (R216).

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* may not appear in a DATA statement.

explicit interface (12.3.1) : For a *procedure referenced* in a *scoping unit*, the property of being an *internal procedure*, a *module procedure*, an *intrinsic procedure*, an *external procedure* that has an *interface block*, a recursive procedure reference in its own scoping unit, or a *dummy procedure* that has an interface block.

explicit-shape array (5.1.2.4.1) : A *named array* that is declared with explicit *bounds*.

expression (2.4.3.2, 7.1) : A sequence of *operands*, *operators*, and parentheses (R723). It may be a *variable*, a *constant*, a *function reference*, or may represent a computation.

extent (2.4.5) : The size of one dimension of an *array*.

external file (9.2.1) : A sequence of *records* that exists in a medium external to the *program*.

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 (9.3) : A mechanism that is used to refer to an *external file*. It is identified by a nonnegative integer.

file (9.2) : An *internal file* or an *external file*.

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 (12.5.2.2) : The *data object* that returns the value of a *function*.

function subprogram (12.5.2.2) : 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*.

global entity (14.1.1) : An *entity* identified by a *lexical token* whose *scope* is a *program*. It may be a *program unit*, a *common block*, or an *external procedure*.

host (2.2.3.2, 2.2.3.3) : A *main program* or *subprogram* that contains an *internal subprogram* is called the **host** of the internal subprogram. A *module* that contains a *module subprogram* is called the **host** of the module subprogram.

host association (14.6.1.3) : The process by which an *internal subprogram*, *module subprogram*, or *derived type* definition accesses *entities* of its *host*.

host scoping unit (2.2) : A *scoping unit* that immediately surrounds another scoping unit.

implicit interface (12.3.1) : A *procedure referenced* in a *scoping unit* other than its own is said to have an implicit interface if the procedure is an *external procedure* that does not have an *interface block*, a *dummy procedure* that does not have an interface block, or a *statement function*.

inquiry function (13.1) : An *intrinsic function* whose result depends on properties of the principal *argument* other than the value of the argument.

intent (12.5.2.1) : An attribute of a *dummy argument* that is neither a *procedure* nor a *pointer*, which indicates whether it is used to transfer data into the procedure, out of the procedure, or both.

instance of a subprogram (12.5.2.4) : The copy of a *subprogram* that is created when a *procedure* defined by the subprogram is *invoked*.

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 (9.2.2) : 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.

intrinsic (2.5.7) : An adjective applied to *data types*, *operations*, *assignment statements*, and *procedures* that are defined in the standard and may be used in any *scoping unit* without further definition or specification.

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) : *Statement keyword* or *argument keyword*.

kind type parameter (2.4.1.1, 4.3.1.1, 4.3.1.2, 4.3.1.3, 4.3.2.1, 4.3.2.2) : A parameter whose values label the available kinds of an *intrinsic type*.

label : See *statement label*.

length of a character string (4.3.2.1) : 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.

literal constant (2.4.3.1.2, 4.3) : A *constant* without a *name*. Note that in FORTRAN 77, this was called simply a constant.

local entity (14.1.2) : An *entity* identified by a *lexical token* whose *scope* is a *scoping unit*.

main program (11.1) : A *program unit* that is not a *module*, *external subprogram*, or *block data program unit*.

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, 11.3) : 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 30 alphanumeric characters (letters, digits, and underscores). Note that in FORTRAN 77, this was called a symbolic name.

name association (14.6.1) : 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 a subobject 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.

nonexecutable statement (2.3.1) : A statement used to configure the program environment in which computational actions take place.

numeric storage unit (14.6.3.1) : 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 (4.3.1) : Integer, real or complex *type*.

object (2.4.3.1) : *Data object*.

obsolescent feature (1.7) : 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 (4.4.1) : When *explicit initialization* or *default initialization* overrides default initialization, it is as if only the overriding initialization were specified.

pointer (2.4.6) : A *variable* that has the POINTER attribute. A pointer shall not be *referenced* or *defined* unless it is *pointer associated* with a *target*. If it is an *array*, it does not have a *shape* unless it is *pointer associated*, although it does have a rank.

pointer assignment (7.5.2) : 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 (7.5.2) : A *statement* of the form "*pointer-object => target*".

pointer associated (6.3, 7.5.2) : The relationship between a *pointer* and a *target* following a *pointer assignment* or a valid execution of an ALLOCATE *statement*.

pointer association (14.6.2) : The process by which a *pointer* becomes *pointer associated* with a *target*.

preconnected (9.3.3) : 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.

present (12.4.1.5) : A *dummy argument* is **present** in an *instance of a subprogram* if it is *associated* with an *actual argument* and the actual argument is a dummy argument that is present in the invoking *subprogram* or is not a dummy argument of the invoking subprogram.

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 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*.

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 (2.5.5) : The appearance of a *data object name* or *subobject 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, 12.5.2.2) : The *variable* that returns the value of a *function*.

scalar (2.4.4) :

(1) A single *datum* that is not an *array*.

(2) Not having the property of being an *array*.

scope (14) : 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 *derived type* definition,

(2) An *interface body*, excluding any derived-type definitions and interface bodies in it, or

(3) A *program unit* or *subprogram*, excluding derived-type definitions, interface bodies, and subprograms in it.

section subscript (6.2.2) : A *subscript*, *vector subscript*, or *subscript triplet* in an *array section selector*.

selector : 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.

shape (2.4.5) : For an *array*, the *rank* and *extents*. The shape may be represented by the rank-one array whose elements are the extents in each dimension.

size (2.4.5) : For an *array*, the total number of elements.

standard-conforming program (1.5) : A *program* that uses only those forms and relationships described in this standard, and which 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 (14) : 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 keyword (2.5.2) : A word that is part of the syntax of a *statement* and that may be used to identify the 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 (14.6.3) : The relationship between two *storage sequences* if a storage unit of one is the same as a storage unit of the other.

storage sequence (14.6.3.1) : A sequence of contiguous *storage units*.

storage unit (14.6.3.1) : A *character storage unit*, a *numeric storage unit*, or an *unspecified storage unit*.

stride (6.2.2.3.1) : The increment specified in a *subscript triplet*.

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* that may be referenced by a *subobject designator*.

subobject (2.4.3.1) : A portion of a *named data object* that may be *referenced* or *defined* independently of other portions. It may be an *array element*, an *array section*, a *structure component*, or a *substring*.

subobject designator (2.5.1) : A *name*, followed by one or more of the following: *component selectors*, *array section* selectors, *array element* selectors, and *substring* selectors.

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 (12.5.2.3) : 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 (5.1.2.8) : A *named data object* specified in a TARGET statement or in a *type declaration statement* containing the TARGET *attribute*, a data object created by an ALLOCATE statement for a pointer, or a *subobject* of such an object.

transformational function (13.1) : An *intrinsic function* that is neither an *elemental* function nor an *inquiry function*. It usually has *array arguments* and an array result whose elements have values that depend on the values of many of the elements of the arguments.

type (4) *:* Data 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 an *intrinsic data type*. KIND and LEN are the type parameters.

type parameter values (4.3) : The values of the *type parameters* of a *data entity* of an *intrinsic data type*.

ultimate component (4.4) : For a *derived type* or a *structure*, a *component* that is of *intrinsic type* or has the POINTER attribute, or an *ultimate component* of a component that is a derived type and does not have the POINTER attribute.

undefined (2.5.4) : For a *data object*, the property of not having a determinate value.

unspecified storage unit (14.6.3.1) : 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 (14.6.1.2) : 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.

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 1.7.1 describes the nature of the deleted features. The Fortran 90 features that are not contained in 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.

(2) Branching to a 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.

(3) PAUSE statement.

The PAUSE statement, present in FORTRAN 66, FORTRAN 77 and for consistency also in Fortran 90, has been deleted.

(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.

(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.

In this and other annexes, FORTRAN 66 is used as the informal name of the first international Fortran standard, ISO 1539:1972, which was technically identical to ANS X3.9-1966.

Recommendations are given in the following sections for those processors which extend the standard by implementing any of the deleted features.

B.1.1 Real and double precision DO variables

Replace rules R821 and R822 in section 8.1.4.1.1 by the following:

"R821 *loop-control* **is** [,] *do-variable* = *scalar-numeric-expr* , ■
■ *scalar-numeric-expr* [, *scalar-numeric-expr*]
or [,] WHILE (*scalar-logical-expr*)

R822 *do-variable* **is** *scalar-variable*

Constraint: The *do-variable* shall be a named *scalar-variable* of type integer, default real, or double precision real.

Constraint: Each *scalar-numeric-expr* in *loop-control* shall be of type integer, default real, or double precision real."

Replace the first part of section 8.1.4.4.1, up to and including the numbered item (1), by the following:

"When the DO statement is executed, the DO construct becomes active. If *loop-control* is

[,] *do-variable* = *scalar-numeric-expr*$_1$, *scalar-numeric-expr*$_2$ [, *scalar-numeric-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 the same type and kind type parameter as the *do-variable*. Their values are established by evaluating *scalar-numeric-expr*$_1$, *scalar-numeric-expr*$_2$, and *scalar-numeric-expr*$_3$, respectively, including, if necessary, conversion to the type and kind type parameter of the *do-variable* according to the rules for numeric conversion (Table 7.10). If *scalar-numeric-expr*$_3$ does not appear, m_3 has the value 1. The value m_3 shall not be zero."

In section 8.1.4.4.1(3), replace "$(m_2 - m_1 + m_3) / m_3$" with "INT($(m_2 - m_1 + m_3) / m_3$)".

Replace rule R901 and the second constraint following it in section 9.4.2 by the following:

"R901 *io-implied-do-control* **is** *do-variable* = *scalar-numeric-expr* , ■
■ *scalar-numeric-expr* [, *scalar-numeric-expr*]

Constraint: The *do-variable* shall be a named scalar variable of type integer, default real, or double precision real.

Constraint: Each *scalar-numeric-expr* in an *io-implied-do-control* shall be of type integer, default real, or double precision real."

B.1.2 Branching to an END IF statement from outside its IF block

In section 8.1.2.2, second paragraph, change the second sentence to be, "It is permissible to branch to an END IF statement from within the IF construct, and also from outside the construct." In section 8.2, change the third paragraph to read, "It is permissible to branch to an END IF statement from within its IF construct, and also from outside the construct."

B.1.3 PAUSE statement

The definition of the statement is:

pause-stmt **is** PAUSE [*stop-code*]

Execution of a PAUSE statement causes a suspension of execution of the program. Execution shall be resumable. At the time of suspension of execution, the stop code, if any, is available in a processor-dependent manner. Leading zero digits in the stop code are not significant. Resumption of execution is not under control of the program. If execution is resumed, the execution sequence continues as though a CONTINUE statement were executed.

For completeness, "**or** *pause-stmt*" should be added to rule R216 in section 2.1.

Constraint: A pure subprogram shall not contain a *pause-stmt*.

B.1.4 ASSIGN, assigned GO TO, and assigned FORMAT

The definitions of the ASSIGN and assigned GO TO statements are:

assign-stmt **is** ASSIGN *label* TO *scalar-int-variable*

Constraint: The label shall be the statement label of a branch target statement or *format-stmt* that appears in the same scoping unit as the *assign-stmt*.

Constraint: *scalar-int-variable* shall be named and of type default integer.

assigned-goto-stmt **is** GO TO *scalar-int-variable* [[,] (*label-list*)]

Constraint: Each label in *label-list* shall be the statement label of a branch target statement that appears in the same scoping unit as the *assigned-goto-stmt*.

Constraint: *scalar-int-variable* shall be named and of type default integer.

Execution of an ASSIGN statement causes a statement label to be assigned to an integer variable. While defined with a statement label value, the integer variable may be referenced only in the context of an assigned GO TO statement or as a format specifier in an input/output statement. An integer variable defined with a statement label value may be redefined with a statement label value or an integer value.

When an input/output statement containing the integer variable as a format specifier (9.4.1.1) is executed, the integer variable shall be defined with the label of a FORMAT statement.

At the time of execution of an assigned GO TO statement, the integer variable shall be defined with the value of a statement label of a branch target statement that appears in the same scoping unit. Note that the variable may be defined with a statement label value only by an ASSIGN statement in the same scoping unit as the assigned GO TO statement.

The execution of the assigned GO TO statement causes a transfer of control so that the branch target statement identified by the statement label currently assigned to the integer variable is executed next.

If the parenthesized list is present, the statement label assigned to the integer variable shall be one of the statement labels in the list. A label may appear more than once in the label list of an assigned GO TO statement.

Further, "*assigned-goto-stm*t" should be added to the lists of prohibited statements in the first constraints to rules R829 and R833 in section 8.1.4.1.2. For completeness, "*assigned-stmt*" and "*assigned-goto-stmt*" should be added to rule R216 in section 2.1.

Add as a list item to the constraint about dummy arguments with INTENT(IN) attribute following R512 in section 5.1.2.3:

(11) In an *assign-stmt*.

In section 14.7.5, the following numbered item should be added: "Execution of an ASSIGN statement causes the variable in the statement to become defined with a statement label value.".

In section 14.7.5, the sentence in numbered item (10), second paragraph, "When a numeric storage unit becomes defined, all associated numeric storage units of the same type become defined" should have the following qualification added at the end, ", except that variables associated with the variable in an ASSIGN statement become undefined when the ASSIGN statement is executed".

In section 14.7.6, the following numbered item should be added: "Execution of an ASSIGN statement causes the variable in the statement to become undefined as an integer. Variables that are associated with the variable also become undefined."

In section 14.7.6, the following numbered item should be added: "A reference to a procedure causes part of a dummy argument to become undefined if the corresponding part of the actual argument is defined with a value that is a statement label value."

In section 12.6, add this item to the constraint that lists prohibited situations in pure subprograms:

(11) In an *assign-stmt*.

In section 9.4.1.1 add to rule R913: "***or*** *scalar-default-int-variable*" with the qualification that the *scalar-default-int-variable* shall have been assigned the statement label of a FORMAT statement that appears in the same scoping unit as the format.

B.1.5 H edit descriptor

In section 10.2.1, add the following line to rule R1016:

" **or** cH *rep-char* [*rep-char*] ..."

Add the following new rule with constraints, which logically follows rule R1016:

" *c* **is** *int-literal-constant*

Constraint: *c* shall be positive.

Constraint: *c* shall not have a kind parameter specified for it.

Constraint: The *rep-char* in the cH form shall be of default character type."

In the H edit descriptor, *c* specifies the number of characters following the H.

If a processor is capable of representing letters in both upper and lower case, the edit descriptors are without regard to case except for the characters following the H in the H edit descriptor and the characters in the character constants.

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 1.7.2 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 non-intuitive restrictions and are a potential source of error since 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 since 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) was an unnecessary redundancy.

Annex C

(informative)

Extended notes

C.1 Section 4 notes

C.1.1 Intrinsic and derived data types (4.3, 4.4)

FORTRAN 77 provided only data 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 data 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 (4.3.1.2).

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 (4.3.2.1).

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.

The distinction between the ultimate and direct components of a derived type can be made clear with an example.

```
TYPE PERSON
  CHARACTER (30) :: NAME
  INTEGER :: AGE
END TYPE PERSON

TYPE HOUSEHOLD
  INTEGER :: NUM_PEOPLE
  TYPE (PERSON), POINTER :: PEOPLE(:)
END TYPE HOUSEHOLD

TYPE COMMUNITY
  INTEGER :: NUM_HOUSES
  TYPE (HOUSEHOLD) :: HOUSES (100)
END TYPE COMMUNITY
```

The ultimate and direct components of PERSON are NAME and AGE. The ultimate and direct components of HOUSEHOLD are NUM_PEOPLE and PEOPLE. The ultimate components of COMMUNITY are NUM_HOUSES, NUM_PEOPLE, and PEOPLE. However, the direct components of COMMUNITY are NUM_HOUSES, HOUSES, NUM_PEOPLE, and PEOPLE. Note that the components of PERSON are neither ultimate nor direct components of HOUSEHOLD or COMMUNITY because the component PEOPLE has the POINTER attribute. The direct components include all the ultimate components and any components of derived type that lead to those ultimate components.

C.1.2 Selection of the approximation methods (4.3.1.2)

This standard permits the selection of the real approximation method for an entire program to be parameterized through the use of the parameterized real data type and module. This is accomplished by defining a named integer constant, say FLOAT, to have a specific kind type parameter value, and to use that named constant in all real, complex, and derived-type declarations. For example, the specification statements

```
INTEGER FLOAT
PARAMETER (FLOAT = 8)
REAL (FLOAT) X, Y
COMPLEX (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 FLOAT can be made available to an entire program by placing the INTEGER and PARAMETER specification statements in a module and accessing the named constant 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^{-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 an exponent range from 10^{-50} to 10^{50}. There are no magnitude or ordering constraints placed on kind values, in order that implementors 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 Pointers (4.4.1)

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 specific object. A normal variable name refers to the same storage space throughout the lifetime of a 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 by its declaration. 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_CELL
   WRITE (*, *) CURRENT % VALUE
END DO
```

C.2 Section 5 notes

C.2.1 The POINTER attribute (5.1.2.7)

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 (5.1.2.8)

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.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 MARY
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.
```

C.3.2 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 currently 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 currently 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)
```

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 should 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 given piece of memory that was allocated and associated with a pointer will become inaccessible to a program if the pointer is nullified 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 (7.5.1.5).

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 (7.1.7.1).

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 currently described by the pointer, that is, the value of the target object currently associated with the pointer. In value-demanding expression contexts, pointers are dereferenced.

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 components are 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,:) .EQ. 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,:) .EQ. 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+1) ) / 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 Additional 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 .NE. 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 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 (9.3)

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 out, 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 (9.2.1.1)

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 (9.3.4.1)

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 (9.2.1.2)

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.6.1.5 Nonadvancing input/output (9.2.1.3.1)

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 currently 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 currently 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 since 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 iolist item was successfully completed. The file will then be positioned after the current record. The IOSTAT= specifier, if present, will be defined with a processor-dependent negative value 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. Implied-DO variables retain their last defined value and any remaining items in the *iolist* 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 I/O in which a nonadvancing write to a terminal device causes immediate display of the output, such a write can be employed 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 I/O 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.6.2 OPEN statement (9.3.4)

A file may become connected to a unit in either of two ways: preconnection or 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 (9.3.3).

The OPEN statement provides a means to access existing files that are not preconnected. An OPEN statement may be used in either of two ways: with a file name (open-by-name) and without a file name (open-by-unit). A unit is given in either case. Open-by-name connects the specified file to the specified unit. Open-by-unit connects a processor-determined default file to the specified unit. (The default file may or may not have a name.)

Therefore, there are three ways a file may become connected and hence processed: preconnection, open-by-name, and open-by-unit. Once a file is connected, there is no means in standard Fortran to determine how it became connected.

An OPEN statement may also be used to create a new file. In fact, any of the foregoing three connection methods may be performed on a file that does not exist. When a unit is preconnected, writing the first record 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 may or may 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=, DELIM=, or PAD= specifiers.

Note that, since an OPEN that specifies STATUS = 'SCRATCH' is not allowed to have a FILE= specifier, such an OPEN always attempts to retain any connection that the specified unit may have. If the unit were already connected to a file, and if that connection did not have a STATUS of SCRATCH, then the OPEN would be illegal because the value of the STATUS= specifier shall not be changed by the OPEN.

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 = 'SCRATCH', 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.6.3 Connection properties (9.3.2)

When a unit becomes connected to a file, either by execution of an OPEN statement or by preconnection, the following connection properties may be established:

(1) An access method, which is sequential or direct, is established for the connection (9.3.4.3).

(2) A form, which is formatted or unformatted, is established for a connection to a file that exists or is created by the connection. For a connection that results from execution of an OPEN statement, a default form (which depends on the access method, as described in 9.2.1.2) is established if no form is specified. For a preconnected file that exists, a form is established by preconnection. For a preconnected file that does not exist, a form may be established, or the establishment of a form may be delayed until the file is created (for example, by execution of a formatted or unformatted WRITE statement) (9.3.4.4).

(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 (9.3.4.5).

(4) A blank significance property, which is ZERO or NULL, is established for a connection for which the form is formatted. This property has no effect on output. For a connection that results from execution of an OPEN statement, the blank significance property is NULL by default if no blank significance property is specified. For a preconnected file, the property is NULL. The blank significance property of the connection is effective at the beginning of each formatted input statement. During execution of the statement, any BN or BZ edit descriptors encountered may temporarily change the effect of embedded and trailing blanks (9.3.4.6).

FORTRAN 77 did not define default values for the blank significance properties of internal and preconnected files. This standard defines the default values for these files to be NULL, matching that of files connected by the OPEN statement.

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 may or may not cause physical activities to take place. It is a place for an implementation to define properties of a file beyond those required in standard Fortran.

C.6.4 CLOSE statement (9.3.5)

Similarly, the actions of dismounting a tape, protection, etc. of a "close" may be implicit at the end of a run. The CLOSE statement may or may not cause such actions to occur. This is another place to extend file properties beyond those of standard Fortran. Note, however, that the execution of a CLOSE statement on 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.6.5 INQUIRE statement (9.6)

Table C.1 indicates the values assigned to the INQUIRE statement specifier variables when no error condition is encountered during execution of the INQUIRE statement.

Table C.1 Values assigned to INQUIRE specifier variables

Specifier	INQUIRE by file		INQUIRE by unit	
	Unconnected	Connected	Connected	Unconnected
ACCESS=	UNDEFINED	SEQUENTIAL or DIRECT		UNDEFINED
ACTION=	UNDEFINED	READ, WRITE, or READWRITE		UNDEFINED
BLANK=	UNDEFINED	NULL, ZERO, or UNDEFINED		UNDEFINED
DELIM=	UNDEFINED	APOSTROPHE, QUOTE, NONE, or UNDEFINED		UNDEFINED
DIRECT=	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN
EXIST=	.TRUE. if file exists, .FALSE. otherwise		.TRUE. if unit exists, .FALSE. otherwise	
FORM=	UNDEFINED	FORMATTED or UNFORMATTED		UNDEFINED
FORMATTED=	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN
IOLENGTH=	RECL= value for *output-item-list*			
IOSTAT=	0 for no error, a positive integer for an error			
NAME=	Filename (may not be same as FILE= value)		Filename if named, else undefined	UNDEFINED
NAMED=	.TRUE.		.TRUE. if file named, .FALSE. otherwise	.FALSE.
NEXTREC=	Undefined	If direct access, next record #; else undefined		Undefined
NUMBER=	–1	Unit number		–1
OPENED=	.FALSE.	.TRUE.		.FALSE.
PAD=	YES	YES or NO		YES
POSITION=	UNDEFINED	REWIND, APPEND, ASIS, UNDEFINED, or a processor-dependent value		UNDEFINED
READ=	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN
READWRITE=	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN
RECL=	Undefined	If direct access, record length; else maximum record length		Undefined
SEQUENTIAL=	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN
UNFORMATTED=	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN
WRITE=	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN

C.7 Section 10 notes

C.7.1 Number of records (10.3, 10.4, 10.6.2)

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 (10.8.1)

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, 11.4)

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 which 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 (11.3)

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 previous standards. Under the previous standards, 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 previous standards, 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 some kind of textual inclusion facility 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 (11.3.2)

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 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 (11.3.1)

The PUBLIC and PRIVATE attributes, which can be declared only in modules, divide the entities in a module into those which 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 on 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 blocks 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 data 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 data type, along with suitable operators.

INTEGER, PARAMETER :: MAX_SET_CARD = 200

TYPE SET                                ! Define SET data 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
```

```
1       END INTERFACE OPERATOR (-)

2       INTERFACE OPERATOR (*)
3          MODULE PROCEDURE INTERSECTION
4       END INTERFACE OPERATOR (*)

5       CONTAINS

6       INTEGER FUNCTION CARDINALITY (A)   ! Returns cardinality of set A
7          TYPE (SET), INTENT (IN) ::  A
8          CARDINALITY = A % CARD
9       END FUNCTION CARDINALITY

10      LOGICAL FUNCTION ELEMENT (X, A)           ! Determines if
11         INTEGER, INTENT(IN) ::  X              ! element X is in set A
12         TYPE (SET), INTENT(IN) ::  A
13         ELEMENT = ANY (A % ELEMENT (1 : A % CARD) .EQ. X)
14      END FUNCTION ELEMENT

15      FUNCTION UNION (A, B)                     ! Union of sets A and B
16         TYPE (SET) UNION
17         TYPE (SET), INTENT(IN) :: A, B
18         INTEGER J
19         UNION = A
20         DO J = 1, B % CARD
21            IF (.NOT. (B % ELEMENT (J) .IN. A)) THEN
22               IF (UNION % CARD < MAX_SET_CARD)  THEN
23                  UNION % CARD = UNION % CARD + 1
24                  UNION % ELEMENT (UNION % CARD) = &
25                        B % ELEMENT (J)
26               ELSE
27                  ! Maximum set size exceeded . . .
28               END IF
29            END IF
30         END DO
31      END FUNCTION UNION

32      FUNCTION DIFFERENCE (A, B)           ! Difference of sets A and B
33         TYPE (SET) DIFFERENCE
34         TYPE (SET), INTENT(IN) :: A, B
35         INTEGER J, X
36         DIFFERENCE % CARD = 0             ! The empty set
37         DO J = 1, A % CARD
38            X = A % ELEMENT (J)
39            IF (.NOT. (X .IN. B)) DIFFERENCE = DIFFERENCE + SET (1, X)
40         END DO
41      END FUNCTION DIFFERENCE

42      FUNCTION INTERSECTION (A, B)       ! Intersection of sets A and B
43         TYPE (SET) INTERSECTION
44         TYPE (SET), INTENT(IN) :: A, B
45         INTERSECTION = A - (A - B)
46      END FUNCTION INTERSECTION

47      LOGICAL FUNCTION SUBSET (A, B)          ! Determines if set A is
48         TYPE (SET), INTENT(IN) ::  A, B      ! a subset of set B
49         INTEGER I
50         SUBSET = A % CARD <= B % CARD
51         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 declaring it in either an EXTERNAL statement or a procedure interface block. 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 in an EXTERNAL statement or a procedure interface block 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-valued 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 a procedure interface block for procedures that do not exist in the program, provided the procedure described is never referenced. The purpose of this rule is to allow implementations in which the use of a module providing procedure interface blocks 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 a procedure interface block would not generate an external reference in such an implementation.)

C.9.4 Argument association and evaluation (12.4.1.1)

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 have been added to 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 which uses only FORTRAN 77 features and which 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.

Note that while this is the specific 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. Alternatively, it would be possible to disallow discontiguous arguments when calling procedures defined by the FORTRAN 77 processor and let any copying to and from contiguous storage be done explicitly in the program. Yet another possibility would be not to allow references to procedures defined by a FORTRAN 77 processor.

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 (7.1.7.1). 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.9.5 Pointers and targets as arguments (12.4.1.1)

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     :: P
    REAL, TARGET      :: 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.10 Section 14 notes

C.10.1 Examples of host association (14.6.1.3)

The first two examples are examples of valid host association. The third example is an example of invalid host association.

Example 1:

```
PROGRAM A
   INTEGER I, J
   ...
CONTAINS
   SUBROUTINE B
      INTEGER I  ! Declaration of I hides
                 ! program A's declaration of I
         ...
      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 shall occur before it is used in the declaration of D (7.1.6.1).

C.11 Array feature notes

C.11.1 Summary of features

This section is a summary of the principal array features.

C.11.1.1 Whole array expressions and assignments (7.5.1.2, 7.5.1.5)

An important new 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 array valued 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 set of lengths in each dimension), but scalars may be included freely and these are interpreted as being broadcast to a conforming array. Expressions are evaluated before any assignment takes place.

C.11.1.2 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.11.1.3 WHERE statement (7.5.3)

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 .GT. 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.11.1.4 Automatic and allocatable arrays (5.1, 5.1.2.4.3)

A major advance for writing modular software is the presence of automatic arrays, created on entry to a subprogram and destroyed on return, and allocatable arrays whose rank is fixed but whose actual size and lifetime is fully under the programmer's control through explicit ALLOCATE and DEALLOCATE statements. The declarations

```
SUBROUTINE X (N, A, B)
   REAL WORK (N, N); 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 allocatable arrays.

C.11.1.5 Array constructors (4.5)

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.11.2 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.11.2.1 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:

A	=	(/ A (1), ..., A (N) /)
X	=	(/ X (1), ..., X (N) /)
COS (X)	=	(/ COS (X (1)), ..., COS (X (N)) /)
A * COS (X)	=	(/ A (1) * COS (X (1)), ..., A (N) * COS (X (N)) /)

The final scalar result is obtained simply by summing the elements of the last of these arrays. Thus, the processor is dealing with arrays at every step of the calculation.

C.11.2.2 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) **.**LT**.** 0.01. Note that the stages of evaluation of this expression are:

A	=	(/ A (1), ..., A (N) /)
ABS (A)	=	(/ ABS (A (1)), ..., ABS (A (N)) /)
ABS (A) **.**LT**.** 0.01	=	(/ ABS (A (1)) **.**LT**.** 0.01, ..., ABS (A (N)) **.**LT**.** 0.01 /)

The conditional Fourier sum we arrive at is:

```
CF = SUM (A * COS (X), MASK = ABS (A) .LT. 0.01)
```

If the mask is all false, the value of CF is zero.

The use of a mask to define a subset of an array is crucial to the action of the WHERE statement. Thus for example, to 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.11.2.3 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 currently have the same parity as it does. Also, the flip is executed only with probability P (4), P (5), or P (6) if as many as 4, 5, or 6 of them have the same parity as it does. (The rule seems to promote neighborhood alignments that may presumably lead to equilibrium in the long run.)

C.11.2.3.1 Problems to be solved

Some of the programming problems that we will need to solve in order to translate the Ising model into Fortran statements using entire arrays are

(1) Counting nearest neighbors that have the same spin;

(2) Providing an array-valued function to return an array of random numbers; and

(3) Determining which gridpoints are to be flipped.

C.11.2.3.2 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-valued 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 FLIP. The decision to flip will be based on the use of uniformly distributed random numbers from the interval $0 \le p < 1$. These will be provided at each gridpoint by the array-valued 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 .EQ. 4) THRESHOLD = P (4)
WHERE (COUNT .EQ. 5) THRESHOLD = P (5)
WHERE (COUNT .EQ. 6) THRESHOLD = P (6)
```

and the spins that are to be flipped are located by the statement:

```
FLIPS = RAND (N) .LE. THRESHOLD
```

All that remains to complete one transition to the next state of the ISING model is to reverse the spins in ISING wherever FLIPS is true:

```
WHERE (FLIPS)  ISING = .NOT. ISING
```

C.11.2.3.3 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 .EQ. 4)  THRESHOLD = P (4)
      WHERE (COUNT .EQ. 5)  THRESHOLD = P (5)
      WHERE (COUNT .EQ. 6)  THRESHOLD = P (6)
      FLIPS = RAND (N) .LE. 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.11.2.3.4 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) .LE. THRESHOLD)  ISING = .NOT. ISING
```

but an extra temporary array would probably be needed. Thus, the scope for saving storage while performing whole array operations is limited. If N is small, this will not matter and the use of whole array operations is likely to lead to good execution speed. If N is large, storage may be very important and adequate efficiency will probably be available by performing the operations plane by plane. The resulting code is not as elegant, but all the arrays except ISING will have size of order N^2 instead of N^3.

C.11.3 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.11.3.1 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 $\begin{bmatrix} B & C & D \\ E & F & G \end{bmatrix}$ the result of this expression is BE + CF + DG.

C.11.3.2 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.11.3.3 Addition of selected elements

The expression $\sum_{x_i > 0.0} x_i$ can be formed using the Fortran expression

```
SUM (X, MASK = X .GT. 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.11.4 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.11.5 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.11.6 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.11.7 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 **.**GT**.** SUM (T) / SIZE (T); N = COUNT (ABOVE); in the example: ABOVE is the logical array (t = true, . = false): $\begin{bmatrix} t & t & t & . \\ t & . & . & t \\ t & . & . & . \end{bmatrix}$ and COUNT (ABOVE) is 6.

Question: What was the lowest score in the above-average group of scores?

Answer: MINVAL (T, MASK = ABOVE), where ABOVE is as defined previously; in 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.11.8 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.11.9 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.11.10 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-valued results.

Annex D
(informative)

Index

The this index, entries in italics denote BNF terms, entries in bold face denote language keywords, and page numbers in bold denote primary or defining text.

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z