X3J3 / S8.104

June 1987

American National Standard for Information Systems Programming Language

Fortran

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

Secretariat: Computer and Business Equipment Manufacturers Association

Draft S8, Version 104
Submitted to X3 by X3J3, Americal National Standards Institute, Inc.

FOREWORD

(This foreword is not part of American National Standard X3.9-198x.)

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

This document is released to X3, the American National Standards Committee for Information Processing Systems, operating under the procedures of the American National Standards Institute, and to SPARC (Standards Planning and Requirements Committee), a subcommittee of X3. The Computer and Business Equipment Manufacturers Association holds the secretariat.

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

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

The committee prefers that the complete capitalization of the language name no longer take place and that the name of the language be spelled "Fortran". Except for referencing FORTRAN 77 and FORTRAN 66, we have used this spelling purposefully in this standard and prefer that all references to the language name in user documentation, industry publications, etc. now use this spelling.

OVERVIEW

Among the additions to FORTRAN 77 in this standard, five stand out as the major ones:

(1) Array operations

(2) Improved facilities for numerical computation

(3) User-defined data types

(4) Facilities for modular data and procedure definitions

(5) The concept of language evolution

A number of other additions are also included in this standard, such as improved source form facilities, more control constructs, recursion, and dynamically allocatable arrays. No FORTRAN 77 features have been deleted.

Array Operations. Computation involving large arrays is an important part of engineering and scientific uses of computing. Arrays may be used as entities in Fortran 8x. Operations for processing whole arrays and subarrays (array sections) are included in the language for two principal reasons: (1) these features provide a more concise and higher level language that will allow programmers more quickly and reliably to develop and maintain scientific/engineering applications, and (2) these features can significantly facilitate optimization of array operations on many computer architectures.

The FORTRAN 77 arithmetic, logical, and character operations and intrinsic functions are extended to operate on array-valued operands. These include whole, partial, and masked array assignment, array-valued constants and expressions, and facilities to define user-supplied array-valued functions. New intrinsic functions are provided to manipulate and construct arrays, to perform gather/scatter operations, and to support extended computational capabilities involving arrays. (For example, an intrinsic function is provided to sum the elements of an array.)

Numerical Computation. Scientific computation is one of the principal application domains of Fortran, and a guiding objective for all of the technical work is to strengthen Fortran as a vehicle for implementing scientific software. Though nonnumeric computations are increasing dramatically in scientific applications, numeric computation remains dominant. Accordingly, the additions include portable control over numeric precision specification, inquiry as to the characteristics of numeric information representation, and improved control of the performance of numerical programs (for example, improved argument range reduction and scaling).

Derived Data Types. "Derived data type" is the term given to that set of features in this standard that allows the programmer to define arbitrary data structures and operations on them. Data structures are user-defined aggregations of intrinsic and derived data types. Intrinsic operations on structured objects include assignment, input/output, and use as procedure arguments. With no additional derived-type operations defined by the user, the derived data type facility is a simple data structuring mechanism. With additional operation definitions, derived data types provide an effective implementation mechanism for data abstractions.

Procedure definitions may be used to define operations on intrinsic or derived data types and nonintrinsic assignments for intrinsic and derived types. These procedures are essentially the same as external procedures, except that they also can be used to define infix operators.

Modular Definitions. In FORTRAN 77, there is no way to define a global data area in only one place and have all the program units in an application use that definition. In addition, the ENTRY statement is awkward and restrictive for implementing a related set of procedures, possibly involving common data objects. Finally, there is no means in FORTRAN 77 by which procedure definitions, especially interface information, may be made known locally to a program unit. These and other deficiencies are remedied by a new type of program unit that may contain any combination of data object declarations, derived data type definitions, procedure definitions, and procedure interface information. This program unit, called a MODULE, may be considered to be a generalization and replacement for the block data program unit. A module may be accessed by any program unit, thereby making the module contents available to that program unit. Thus, modules provide improved facilities for defining global data areas, procedure packages, and encapsulated data abstractions.

Language Evolution. With the addition of new facilities, certain old features become redundant and may eventually be phased out of the language as use declines. For example, the numeric facilities alluded to above provide the functionality of double precision; with the new array facilities, nonconformable argument association (such as associating an array element with a dummy array) is unnecessary (and in fact is not useful as an array operation); and

BLOCK DATA program units are redundant and inferior to modules.

As part of the evolution of the language, categories of language features (deleted, obsolescent, and deprecated) are provided which allow unused features of the language to be removed from future standards.

DOCUMENT ORGANIZATION

This document 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 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 an executable program and the relationships of its major parts. Also included are the syntax of program units, the rules for statement ordering, and the definition of many of the fundamental terms used throughout the document.

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 (arrays as data objects) 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 (predefined) data type is introduced, and the properties of Fortran's intrinsic types (INTEGER, REAL, including specified precision REAL and DOUBLE PRECISION, 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 resolve into intrinsic types. The details for defining a derived type are given (note that this has no counterpart with intrinsic types as 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 object's data type, so that the type declaration statement is the main feature of this section. The different 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. Arrays are considered to be variables, as are array sections. Among the array facilities described here are array sections (subarrays), array allocation and deallocation (user controlled dynamic arrays), effective array ranges, range control (SET RANGE), and dynamic aliasing for arrays (IDENTIFY). The section concludes with a summary of the allowed appearances of array names.

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 two principal forms: the conventional assignment statement and the WHERE statement and construct. The WHERE statement and construct allow masked array assignment.

Section 13 (Intrinsic Procedures) describes the approximately one hundred intrinsic functions and four intrinsic subroutines of Fortran that provide a rich set of computational capabilities. In addition to the FORTRAN 77 intrinsics, this includes many array processing functions and a comprehensive set of numerical environmental inquiry functions.

Execution Control. Section 8 (Execution Control) describes the control constructs (IF, SELECT CASE, and DO), branching statements (various forms of GO TO), and other control statements (for example, logical IF, arithmetic IF, CONTINUE, STOP, and PAUSE). These are as in FORTRAN 77 except for the addition of the SELECT CASE construct and extension of the DO loop to include an END DO termination option, additional control clauses, and addition of EXIT and CYCLE statements.

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), file positioning, 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 I/O, and namelist I/O.

Program Units. Section 11 (Program Units) describes main programs, 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.

Section 12 also describes the use of the OPERATOR option on function definitions 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 on subroutine definitions 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, overloaded procedure names, and

non-Fortran procedures.

Scoping and Association Rules. Section 14 (Scope, Association, and Definition) explains the use of the term "scope" (especially important now because of the addition of internal procedures, modules, and other new features), and describes the scope properties of various entities, including names, operators, and others. Also described are the general rules governing procedure argument association, use association (accessing entities in modules), 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.

X3J3 COMMITTEE

Administration of X3J3 has been undertaken by a "Steering Committee" and the technical development has been carried out by subgroups, whose work is reviewed by the full committee. During the period of development of the draft Fortran standard, many persons assumed important roles of leadership. Their contributions are mentioned on the following page. At the present time, the membership consists of 37 members.

STEERING COMMITTEE

Jeanne C. Adams, Chair
Jerrold L. Wagener, Vice-Chair
Walter S. Brainerd, Director, Technical Work
Lloyd W. Campbell, Editor
Jeanne T. Martin, Secretary
Neldon H. Marshall, Librarian
E. Andrew Johnson, Interpretations
James H. Matheny, Vocabulary Representative

SUBGROUP HEADS	(Assistant Heads)
Richard A. Hendrickson	(Alan Wilson)
Kurt W. Hirchert	(John K. Reid)
James H. Matheny	(Murray F. Freeman)
Richard R. Ragan	(J. Lawrence Schonfelder)
E. Andrew Johnson	(Jerrold L. Wagener)

Subcommittee X3J3 on Fortran, with the guidance of the international Fortran Working Group ISO/TC97/SC22/WG5, developed this standard. Those who contributed to the work by attending four or more meetings were:

Jeanne C. Adams, Chair
Jerrold L. Wagener, Vice-Chair
Martin N. Greenfield, Vice-Chair (1972-1985)
Walter S. Brainerd, Director, Technical Work*
Lloyd W. Campbell, Editor*
Jeanne T. Martin, Secretary*
Loren P. Meissner, Secretary (1978-1982)
Vacant, International Representative
Frances E. Holberton, International Representative (1978-1982)
Neldon H. Marshall, Librarian*
James H. Matheny, Vocabulary Representative*
Jeanne T. Martin, Convener ISO/TC97/SC22/WG5

Cornelis G. F. Ampt
Stuart L. Anderson
Charles Arnold
Graham Barber
Gloria M. Bauer*
Valerie G. Bowe
Joanne Brixius
Neil Brutman
Albert Buckley
Larry Bumgarner
Carl D. Burch
Winfried A. Burke*
John H. Carman
T. C. Chao
Nancy Cheng
P. Alan Clarke
Joel Clinkenbeard
Joe Cointment
Theodore R. Crowley
Ingemar Dahlstrand
Chela Diaz de Villegas
David C. Dillon
Joe L. Dowdell
T. Miles R. Ellis
John T. Engle
Stuart I. Feldman
Francoise Ficheux-Vapne
Murray F. Freeman
Daniel A. Gallagher
Gary L. Graunke
Stephen R. Greenwood
Richard B. Grove*
Kevin W. Harris
Richard A. Hendrickson*
Dean A. Herington*
Kurt W. Hirchert*
Tracy Ann Hoover
Sheryl Horowitz
Steve K. Hue
Jagmohan L. Humar
E. Andrew Johnson*
Gregory Johnson
Peter N. Karculias
Leslie M. Klein
Wilfried Kneis
Werner Koblitz
George T. Komorowski
Joseph A. Korty
Anil K. Lakhwara
Dorothy E. Lang
John E. Lauer*
Kay Leonard
Donald L. Loe
Warren E. Loper
Bruce A. Martin*
Alex L. Marusak
Christian J. Mas
John Mayer
Edward H. McCall
Brian L. Meek
Michael Metcalf
Geoff Millard
Robert M. Miller
Leonard J. Moss
Meinolf Munchhausen
David T. Muxworthy
Linda J. O'Gara
Rod R. Oldehoeft
John P. Olson*
Rex L. Page*
George Paul
Daniel Pearl
Odd Pettersen
Ivor R. Philips
Aurelio A. Pollicini
Bruce W. Puerling*
Richard R. Ragan*
John K. Reid
Lawrence Rolison
Karl-Heinz Rotthauser
Steven M. Rowan
Werner Schenk*
Gerhard J. Schmitt
J. Lawrence Schonfelder
Rick N. Schubert
John C. Schwebel
Mok-Kong Shen
Richard Shepardson
Richard W. Signor*
Brian T. Smith*
Jan A. M. Snoek
Hieronymus Sobiesiak
Ken Sperka
Bruce Stowell
Sylvia Sund
Mario Surdi
Richard C. Swift
Brian L. Thompson
Christian Ullrich
Robert B. Upshaw*
Nico Vossenstijn
Richard W. Weaver
George E. Weekly
Bruce Weinman
Everett H. Whitley
Gunter Wiesner
Edward J. Wilkens
Alan Wilson
John D. Wilson

*Subgroup Head

TABLE OF CONTENTS

FIGURES

TABLES

1. INTRODUCTION

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

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

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

1.3.1. Inclusions. This standard specifies:

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

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

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

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

1.3.2. Exclusions. This standard does not specify:

(1) The mechanism by which programs are transformed for use on 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 the rules of this standard fail to establish an interpretation

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

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

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

(8) The physical properties and implementation of storage

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

An executable program (2.2.2) conforms to this standard if it uses only those forms and relationships described herein and if the executable program has an interpretation according to this standard. A program unit (2.2) conforms to this standard if it can be included in an executable program in a manner that allows the executable program to be standard conforming.

A processor conforms to this standard if:

(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 deleted, obsolescent, or deprecated, 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.

However, a processor is not required to detect or report the use of deleted, obsolescent, or deprecated features, nor the use of additional forms or relationships occurring in a *format-specification* that is not part of a *format-stmt* (10.1.1).

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 appears in an EXTERNAL statement. A standard-conforming program must not use nonstandard intrinsic procedures that have been added by the processor.

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

Note that a standard-conforming program must not use any forms or relationships that are prohibited by this standard, but a standard-conforming processor may allow such forms and relationships if they do not change the proper interpretation of a standard-conforming program. For example, a standard-conforming processor may allow a nonstandard data type such as POINTER.

Because a standard-conforming program may place demands on a processor that are not within the scope of this standard or may include standard items that are not portable, such as external procedures defined by means other than Fortran, conformance to this standard does not ensure that a standard-conforming program will execute consistently on all or any standard-conforming processors.

1.5. Notation Used in This Standard.

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

1.5.1. Syntax Rules.

Syntax rules are used to help describe the form that Fortran 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 are to be written as shown, except where otherwise noted.

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

Some common abbreviations used in syntactic terms are:

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

(3) The syntactic metasymbols used are:

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

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

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

(6) Obsolescent features (1.6) are shown in a distinguishing type font. This is an example of the font used for obsolescent features.

An example of the use of syntax rules is:

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

The following forms are examples of forms for an integer constant 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 integer constants might be:

4
67
1999
10243852

1.5.2. Assumed Syntax Rules. 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* must be scalar.

1.5.3. Syntax Conventions and Characteristics.

(1) Any syntactic class name ending in "*-stmt*" follows the source form statement rules: it must 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-R223). Expression hierarchy is described rigorously in the definition of *expr* (R712).

(3) The term "type parameter" applies to a data type parameter, with "*type-param-name*" used for the dummy parameter and "*type-param-spec*" (R503) used for the actual parameter, including the optional keyword. The part without the keyword is called "*type-param-value*" (R504). These terms parallel the use of "*dummy-arg-name*", "*actual-arg-spec*" (R1209) and "*actual-arg*" (R1211), respectively, for procedure arguments.

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

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

(6) The term "*subscript*" (e.g., R614 and R617) is used consistently in array definitions.

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

1.6. Deleted, Obsolescent, and Deprecated Features.

This standard protects the users' investment in existing software by including all of the language elements of ANSI X3.9-1978. This document identifies three categories of outmoded features. There are none in the first category, **deleted features**, which consists of features considered to have been redundant and largely unused in ANSI X3.9-1978. Those in the second category, **obsolescent features**, are considered to have been redundant in ANSI X3.9-1978, but are still used frequently. Those in the third category, **deprecated features**, are considered to have become redundant by the inclusion of certain new features in this standard. Sections 1.6.1 and 1.6.2 describe the first two categories; Appendix B describes the third and lists the features in each.

1.6.1. Nature of Deleted Features.

(1) Better methods existed in ANSI X3.9-1978.

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

1.6.2. Nature of Obsolescent Features.

(1) Better methods existed in ANSI X3.9-1978.

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

(3) These features are identified in the text of this document by a distinguishing type font (1.5.1).

(4) If the use of these features has become insignificant in Fortran programs, it is recommended that future Fortran standards committees consider removing them from the next revision.

(5) It is recommended that future Fortran standards committees do not consider removing language features defined in this revision from the succeeding Fortran revision that do not appear on the list of obsolescent features.

(6) It is recommended that processors supporting the Fortran language continue to support these features as long as they continue to be used widely in Fortran programs.

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

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

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 syntax rule notation is described in 1.5.

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

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

R202 *program-unit* **is** *main-program*
or *external-subprogram*
or *module*
or *block-data*

R203 *main-program* **is** [*program-stmt*]
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-program-stmt

R204 *external-subprogram* **is** *procedure-heading*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
procedure-ending

R205 *procedure-heading* **is** *function-stmt*
or *subroutine-stmt*

R206 *procedure-ending* **is** *end-function-stmt*
or *end-subroutine-stmt*

Constraint: In an *external-subprogram*, *module-subprogram*, or *internal-subprogram*, the *procedure-ending* must be *end-function-stmt* if the *procedure-heading* is a *function-stmt* and must be *end-subroutine-stmt* if the *procedure-heading* is *subroutine-stmt*.

R207 *module* **is** *module-stmt*
[*specification-part*]
[*module-subprogram-part*]
end-module-stmt

Constraint: A module *specification-part* must not contain a *stmt-function-stmt*, an *entry-stmt*, a *format-stmt*, an *intent-stmt*, an INTENT attribute, an *optional-stmt*, or an OPTIONAL attribute.

R208 *block-data* **is** *block-data-stmt*
[*specification-part*]
end-block-data-stmt

Constraint: A *block-data specification-part* may contain only IMPLICIT, PARAMETER, INTEGER, REAL, DOUBLE PRECISION, COMPLEX, CHARACTER, LOGICAL, COMMON, DIMENSION, EQUIVALENCE, DATA, and SAVE statements.

R209 *specification-part* **is** [*use-stmt*]...
[*implicit-part*]
[*declaration-construct*]...
[*stmt-function-part*]

R210 *implicit-part* **is** [*implicit-part-stmt*]...
implicit-stmt

R211 *stmt-function-part* **is** *stmt-function-stmt*
[*stmt-function-part-stmt*]...

R212 *implicit-part-stmt* **is** *implicit-stmt*
or *parameter-stmt*
or *format-stmt*
or *entry-stmt*

R213 *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*

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

R215 *execution-part* **is** *executable-construct*
[*execution-part-construct*]...

R216 *execution-part-construct* **is** *executable-construct*
or *format-stmt*
or *data-stmt*
or *entry-stmt*

R217 *internal-subprogram-part* **is** *contains-stmt*
[*internal-subprogram*]...

R218 *internal-subprogram* **is** *procedure-heading*
[*specification-part*]
[*execution-part*]
procedure-ending

R219 *module-subprogram-part* **is** *contains-stmt*
[*module-subprogram*]...

R220 *module-subprogram* **is** *procedure-heading*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
procedure-ending

R221 *specification-stmt* **is** *access-stmt*
or *data-stmt*
or *exponent-letter-stmt*
or *external-stmt*
or *intent-stmt*
or *intrinsic-stmt*
or *namelist-stmt*
or *optional-stmt*
or *range-stmt*
or *save-stmt*
or *common-stmt*
or *dimension-stmt*
or *equivalence-stmt*

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

Constraint: An *intent-stmt* or *optional-stmt* may appear only in the *specification-part* of a subprogram or the *declaration-construct* of an interface block because they apply only to dummy arguments.

R222 *executable-construct* **is** *action-stmt*
or *case-construct*
or *do-construct*
or *if-construct*
or *where-construct*

R223 *action-stmt* **is** *allocate-stmt*
or *assignment-stmt*
or *backspace-stmt*
or *call-stmt*
or *close-stmt*
or *computed-goto-stmt*
or *continue-stmt*
or *cycle-stmt*
or *deallocate-stmt*
or *endfile-stmt*
or *exit-stmt*
or *goto-stmt*
or *identify-stmt*
or *if-stmt*
or *inquire-stmt*
or *open-stmt*
or *print-stmt*
or *read-stmt*
or *return-stmt*
or *rewind-stmt*
or *set-range-stmt*
or *stop-stmt*
or *where-stmt*
or *write-stmt*
or arithmetic-if-stmt
or assign-stmt
or assigned-goto-stmt
or pause-stmt

Constraint: An *entry-stmt* may appear only in an *external-subprogram* or *module-subprogram*. An *entry-stmt* must not appear within an executable construct.

Constraint: A *return-stmt* may appear only in a subprogram.

Constraint: An *exit-stmt* or a *cycle-stmt* may appear only in a *do-construct*.

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 named common block data objects. Each type of program unit is described in Section 11 or 12. An **external subprogram** is a subprogram that is not contained within a main program, a module, or another subprogram. An **internal subprogram** is a subprogram that is contained within a main program or another subprogram. A **module subprogram** is a subprogram that is contained in a

module but is not an internal subprogram.

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

(1) A derived-type definition,

(2) A procedure interface block, excluding any procedure interface blocks contained within it, or

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

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

2.2.2. Executable Program. An **executable 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.

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

2.2.4. Procedure. A **procedure** encapsulates arbitrary computations that may be invoked directly during program execution. A principal difference between the two kinds of procedures is the way in which each is invoked. 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. A **subroutine** is a procedure that is invoked in a CALL statement or by an assignment operation (12.4.4, 12.5.2.3). A subroutine may be used to change the program state by changing the values of any of the data objects accessible to the subroutine; a function subprogram may do this in addition to computing the function value.

Procedures are described further in Section 12.

2.2.4.1. 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 any procedure of an executable program.

2.2.4.2. Module Procedure. A **module procedure** is a procedure that is defined by a module subprogram (R220). A module procedure may be invoked by another module subprogram in the module or by any program unit using the module. The module containing the subprogram is called the **host** of the module procedure.

2.2.4.3. Internal Procedure. An **internal procedure** is a procedure that is defined by an internal subprogram. 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.

2.2.4.4. Procedure Interface Block. The purpose of a **procedure interface block** is to describe the interface (12.3) to a procedure. It determines the forms of reference through which the procedure may be invoked.

2.2.5. 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 in an executable program that request such accessibility. A scoping unit in another program unit may request access to the definitions contained in a module. Modules are further described in

Section 11.

2.3. Execution Concepts. A program unit is a sequence of statements. Each 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 must be nonexecutable. A module may contain executable statements only within a subprogram in the module.

2.3.2. Statement Order. The syntax rules of Section 2.1 specify the statement order within program units and subprograms. These rules are illustrated in Figure 2.1 and Table 2.1. Figure 2.1 applies to all program units and subprograms and shows the ordering rules for statements. Table 2.1 shows which statements are allowed in the scoping unit of a program unit, a subprogram, or an interface block (12.3.2.1). In Figure 2.1, vertical lines delineate varieties of statements that may be interspersed and horizontal lines delineate varieties of statements that must not be interspersed. USE statements, if any, must appear immediately after the program unit heading. Internal procedure definitions or module subprograms must follow a CONTAINS statement. Between USE statements and internal procedure definitions, nonexecutable statements generally precede executable statements, though the FORMAT statement, DATA statement, and ENTRY statement may appear among the executable statements.

Figure 2.1. Requirements on Statement Ordering.

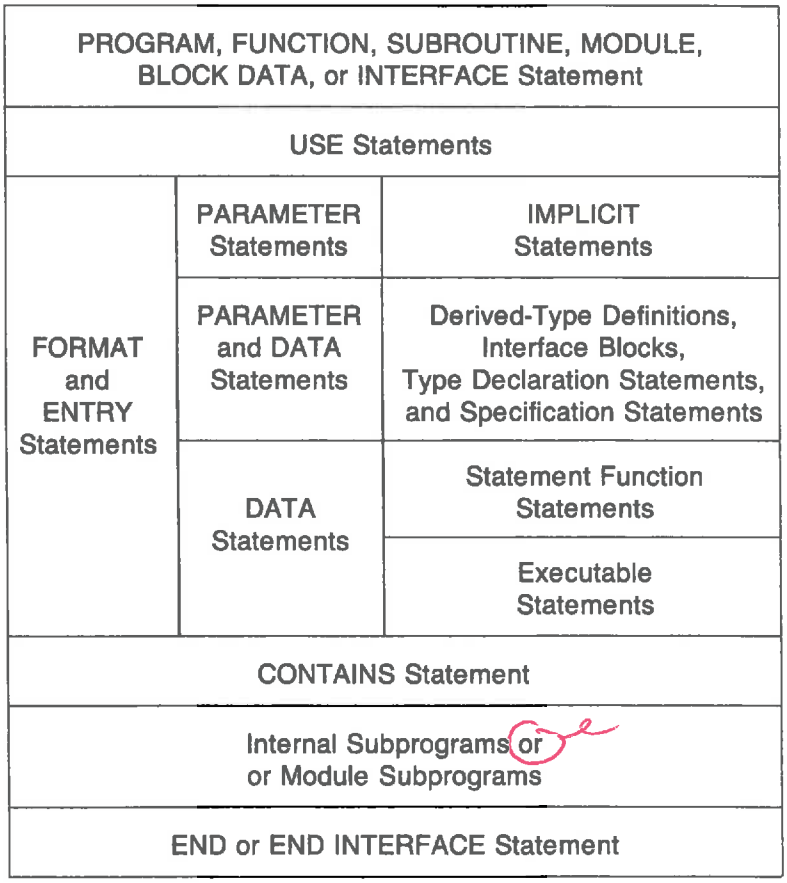


Table 2.1. Statements Allowed in Scoping Units.

Kind of program unit:	Main Program	Module	Block Data	External Subprog	Module Subprog	Internal Subprog	Interface Block
USE Statement	Yes	Yes	No	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
Derived Type Definition	Yes	Yes	No	Yes	Yes	Yes	Yes
Interface Blocks	Yes	Yes	No	Yes	Yes	Yes	Yes
Statement Function	Yes	Yes	No	Yes	Yes	Yes	No
Executable Statement	Yes	No	No	Yes	Yes	Yes	No
CONTAINS	Yes	Yes	No	Yes	Yes	No	No

Note: Misc. Declarations are PARAMETER Statements, IMPLICIT Statements, DATA Statements, Type Declarations, and Specification Statements.

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 and subprogram must have exactly one END statement, which may be labeled; it must be the last statement of the program unit or subprogram. 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* is equivalent to executing a *stop-stmt* in a main 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. The execution of a main program or subprogram involves execution of the executable constructs of its scoping unit. Upon invocation of a procedure, execution begins with the first executable construct appearing after the invoked entry point. With the following exceptions, the executable constructs are executed in the order in which they appear in the main program or subprogram until a STOP, RETURN, or END statement is executed. The exceptions are:

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

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

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

(4) Internal subprograms may precede the END statement of a main program or a subprogram. The execution sequence skips 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. A 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**.

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 implicitly defined, 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.

2.4.1.2. Derived Type. A **derived type** is a type that is not implicitly defined but requires a type definition that contains component declarations to declare components of intrinsic or of other derived types. Derived types are characterized by a small set of intrinsic operations: assignment with type agreement, use as procedure arguments and function results, inquiry functions for type parameter values, and input/output. If additional operations are needed for a derived type, they must be supplied as procedure definitions.

Intrinsic types are accessible to every scoping unit. A derived-type definition is local to the scoping unit in which it appears, but may be accessed from other scoping units by use or host association (14.7.1, 11.3.1).

Derived types are described further in 4.4.

2.4.2. Data Value. Each intrinsic type has associated with it a set of intrinsic values that 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 intrinsic values.

2.4.3. Data Entity. A **data entity** is an entity that has, or may have, a data value. A data entity is a constant, a variable, an expression value, or a function result. In addition, it is either a scalar or an array.

2.4.3.1. Data Object. A **data object** (often abbreviated to **object**) is a datum or set of data of the same type and type parameters that may be referenced as a whole.

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

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

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

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

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

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

2.4.7. Array. An **array** is a set of data, all of the same type and type parameters, whose individual elements are arranged in a rectangular pattern. An **array element** is one of the individual elements in the array and is a scalar. An **array section** is a subset of the elements of an array and is itself an array.

An array with a name has one subscript for each dimension of the pattern. The pattern may have up to seven dimensions, and any **extent** (size) in any dimension. The **rank** of the array is the number of dimensions, and its **size** is the total number of elements, which is equal to the product of the extents. Arrays may have zero size. The **shape** of an array is determined by its rank and its extent in each dimension, and may be represented as a rank-one array whose elements are the extents. The rank of a scalar is zero. All named arrays must 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 dummy argument arrays, automatic arrays, alias arrays, ranged arrays, and allocatable arrays.

Two arrays are said to be **conformable** if they have the same effective shape. A scalar is deemed to be conformable with any array. Any 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 simultaneously.

A rank-one array may be constructed from scalars and other rank-one arrays and may be reshaped into any allowable array shape (4.5).

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

2.4.8. 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 must observe certain storage constraints (14.7.2).

There are two kinds of physical **storage units**: numeric and character. When used in a storage association context, scalar objects of type integer, default real, and logical each use a single numeric storage unit. When used in a storage association context, scalar objects of type double precision real and default complex each use two contiguous numeric storage units. When used in a storage association context, each character in an object of type character uses one character storage unit and scalar character objects employ a contiguous set of such units. When used in a storage association context, array objects are assigned contiguous storage units of the appropriate kind, in array element order (6.2.4.2). For example, the storage order for a two-dimensional array is the first column followed by the second column, etc.

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

2.5. Fundamental Terms.

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

2.5.1. 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 3.2.2. A **subobject designator** is a name followed by one or more 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 and that may be used to identify the statement is a **statement keyword**. Examples of this kind of keyword are: IF, READ, WHERE, and INTEGER. These keywords are not "reserved words"; that is, names with the same spellings are allowed.

An argument keyword is a dummy argument name. Section 13 defines 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).

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 ceases to have a predictable value and is said to become **undefined**. Section 14 describes the ways in which variables may become defined and undefined. The second use of the term definition is for the definition of derived types and procedures.

2.5.5. Reference. A **data object reference** is the appearance of the data object name or 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 needed to complete the specification of the actual

argument.

2.5.6. Association. An **association** exists if an entity may be identified by different names in the same scoping unit or by the same name or different names in different scoping units. It may be name association (14.7.1) or storage association (14.7.2). Name association may be argument association, host association, use association, or alias association.

2.5.7. Intrinsic. The term **intrinsic** applies to intrinsic data types, intrinsic procedures, and intrinsic operators that are defined in this standard. These 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 also may be defined.

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

R301 *character* **is** *alphanumeric-character*
or *special-character*

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

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

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

The twenty-six letters define 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 character constants, delimited character edit descriptors, and H edit descriptors.

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

0 1 2 3 4 5 6 7 8 9

The ten digits define the syntactic class *digit*. When used in numeric constants, the digits are interpreted according to the decimal base number system.

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-three **special characters** are:

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

The twenty-three special characters define the syntactic class *special-character*. The special characters are used for operator symbols, bracketing, and various forms of separating and delimiting of other lexical tokens. The special characters $ and ? have no specified use. Where brackets are used, alternate syntax is permitted. The left bracket may be replaced by

the character pair (/. The right bracket may be replaced by the character pair /).

3.1.5. Character Graphics. Except for the currency symbol, the graphics used for the characters must 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.

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

ICHAR (*character*)

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

The only constraints on the collating sequence are:

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

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

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

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

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

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

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

3.2. Low-Level Syntax.

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

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

3.2.2. 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.

Examples of names:

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

3.2.3. 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*

R307 *named-constant* **is** *name*

R308 *int-constant* **is** *constant*

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

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

Constraint: *char-constant* must be of type character.

3.2.4. 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*

R311 *power-op* **is** **

R312 *mult-op* **is** *
or /

R313 *add-op* **is** +
or -

R314 *concat-op* **is** //

R315 *rel-op* **is** .EQ.
or .NE.
or .LT.
or .LE.
or .GT.
or .GE.
or = =
or < >
or <
or < =
or >
or > =

R316 *not-op* **is** .NOT.

R317 *and-op* **is** .AND.

R318 *or-op* **is** .OR.

R319 *equiv-op* **is** .EQV.

or .NEQV.

R320 *defined-operator* **is** *defined-unary-op*
or *defined-binary-op*
or *overloaded-intrinsic-op*

R321 *defined-unary-op* **is** . *letter* [*letter*]... .

R322 *defined-binary-op* **is** . *letter* [*letter*]... .

R323 *overloaded-intrinsic-op* **is** *intrinsic-operator*

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

3.2.5. Statement Labels. Any statement not forming part of another statement may be labeled.

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

Constraint: At least one digit in the label must be nonzero.

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

```
99999
10
1 0
010
```

are all statement labels. The last three are equivalent.

3.2.6. Delimiters. Many of the special characters are used as delimiters, as described in the syntax rules.

3.3. Source Form.

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

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

A **character context** means characters within (between the delimiters for) a character constant, a format-item in a FORMAT statement, and a comment. Any character representable in the processor may occur in a character context.

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

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

3.3.1. Free Source Form. In free form, each source record may contain from zero to a maximum of 132 characters.

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

3.3.1.2. Statement Separation. The character ";" separates statements on a single source line except when it appears within a character context. Statements containing no characters or only blanks are ignored.

3.3.1.3. Statement Continuation. Outside of a comment, the character "&" as the last nonblank character on a line signifies that the statement is continued on the next line. A comment cannot be continued. A "&" in a character context that is a comment has no effect. If the first nonblank character on the next line is also "&", the statement continues at the next character position following the "&"; otherwise, it continues at character position 1. When used for continuation, the "&" is not part of the statement. If a character context other than a comment is being continued, the "&" signifying continuation cannot be followed by commentary and the continued portion must begin with an "&". If the continuation is not within a character context, the "&" signifying continuation may be followed by commentary. A statement must not contain more than 2640 characters.

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

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

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

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

(4) An "!" in character position 6 does not initiate a comment and indicates a continuation line except within a comment.

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

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

4. INTRINSIC AND DERIVED DATA TYPES

Fortran provides an abstract means that permits the categorization of data 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 defined by the language; the names of any derived types must 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.

For example, the logical data type has a set of two values, denoted by the 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 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. Some data types may be parameterized. In this case, the set of values is constrained by 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. An intrinsic type is always accessible.

In addition to the intrinsic types, application specific types may be derived. Derived types have **components**. Each component is of an intrinsic type or of another 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. For example, if the complex data 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.

Means are provided to denote values of a derived type (4.4.3) and to define operations that can be used to manipulate objects of a derived type (4.4.4). A derived type must be completely defined, whereas an intrinsic type is predefined.

4.1. The Concept of Type.

A data type has (1) a name, (2) a set of valid values, (3) a means to denote such values (constants), and (4) a set of operations to manipulate the values. A type may be parameterized, in which case the set of values is determined by the values of the parameters.

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 form for literal constants of that type is specified in this standard. These literal constants are described in 4.3 for each intrinsic type.

A constant value may be given a name.

A constant value of derived type may be constructed (4.4.3) using a derived-type 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.

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 operator functions (12.5.2.2). Operator definitions are described in Sections 7 and 12.

For derived types, the only intrinsic operation is assignment. All other operations must be defined.

4.2. Relationship of Types and Values to Objects and Entities.

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 attribute for 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. An array is an object and has a type just as a scalar object does. Thus data objects, such as arrays, may be collections of subobjects.

An object of derived type is referred to as a **structure** or a **structured object** with components. The components of a structured object 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 effective 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 effective shapes is described in Section 7. For example, assignment of an integer value to a real variable is intrinsically defined. For an assignment that is not intrinsically defined, conversions may be defined by assignment subroutines (Section 7 and 12.5.2.3).

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

A **data entity** is an entity that has or may have a data value. It may be a constant, a variable, an expression value, or a function result. Each data entity has a data type. The type may be specified (if the entity is a variable or function result) or it may be determined by the rules in Section 7 (if the entity is an expression value).

4.3. Intrinsic Data Types.

The intrinsic data types are:

numeric types:	Integer, Real, Complex
nonnumeric types:	Character and Logical

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.

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

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

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

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

R401 *signed-int-literal-constant* **is** [*sign*] *int-literal-constant*

R402 *int-literal-constant* **is** *digit* [*digit*]...

R403 *sign* **is** +
or –

Examples of unsigned and signed integer literal constants are:

```
473
5 000 000
+56
-101
```

An integer constant is interpreted as a decimal value.

4.3.1.2 Real and Double Precision Real Type. The real type has values that approximate the mathematical real numbers. A processor must provide two or more **approximation methods** that define sets of values for data of type real. Each such method is characterized by an effective decimal precision and an effective decimal exponent range. The effective decimal precision of an approximation method is returned by the intrinsic inquiry function EFFECTIVE_PRECISION (13.12.33) and the effective decimal range is returned by the intrinsic inquiry function EFFECTIVE_EXPONENT_RANGE (13.12.32).

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

If one of the type parameters is omitted in the specification of a data entity of type real, a processor-dependent default is used. The type parameters of such an entity are regarded as different from those of any entity for which both parameters are specified.

If neither type parameter is specified, a processor-defined default real approximation method is selected and the data entity is of type **default real**. The type parameters of such an entity are regarded as different from those of any entity for which one or both parameters are specified.

If double precision is specified for a data entity, a processor-defined double precision approximation method is selected and the entity is of type **double precision real**. The type parameters of such an entity are regarded as different from those of any entity of type real for which one or both parameters are specified. The effective decimal precision of the double precision approximation method must be greater than that of the default real method.

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.

R404 *signed-real-literal-constant* **is** [*sign*] *real-literal-constant*

R405 *real-literal-constant* **is** *significand* [*exponent-letter exponent*]
or *int-literal-constant exponent-letter exponent*

R406 *significand* **is** *int-literal-constant* **.** [*int-literal-constant*]
or **.** *int-literal-constant*

R407 *exponent* **is** *signed-int-literal-constant*

R408 *exponent-letter* **is** E

or D
or *defined-exponent-letter*

R409 *exponent-letter-stmt* **is** EXPONENT LETTER *precision-selector* ■
■ *defined-exponent-letter*

R410 *defined-exponent-letter* **is** *letter*

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

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

Real literal constants written without an exponent part, or with exponent letter E, are default real objects; exponent letter D specifies a double precision real constant. A specified precision real constant must use the exponent character specified for that precision in an EXPONENT LETTER statement in the same scoping unit.

Examples of signed real literal constants are:

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

Examples of unsigned real literal constants are:

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

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

The EXPONENT LETTER statement must be used if a numeric constant is to be passed to a dummy argument of the type real with specified precision and exponent range. An example is:

```
EXPONENT LETTER (10,50) L
...
CALL SUB (1.2LO)
...
SUBROUTINE SUB (DUMMY)
REAL (10,50) DUMMY
```

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

4.3.1.3 Complex Type. The **complex type** 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**.

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

If neither the precision nor the exponent range is specified, the default real method is selected for both parts and the complex data entity is **default complex**.

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

R411 *complex-literal-constant* **is** (*real-part* , *imag-part*)

R412 *real-part* **is** *signed-int-literal-constant*
or *signed-real-literal-constant*

R413 *imag-part* **is** *signed-int-literal-constant*
or *signed-real-literal-constant*

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

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

Examples of complex literal constants are:

```
(1.0, -1.0)
(3, 3.1E6)
```

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

4.3.2.1 Character Type. The **character type** 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. Any character representable in the processor may occur in a character string. Strings of different lengths are all of type character.

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

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

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

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

The intrinsic operation **concatenation** (//) is defined between two data entities of type character (7.2.2).

Examples of character literal constants are:

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

both of which have the value DON'T.

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

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

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 logical value.

These operations are described in 7.2.3.

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

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. A component may be declared to be of any intrinsic or previously defined derived type. Ultimately, a derived type is resolved into a sequence of components of intrinsic type.

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

4.4.1 Derived-Type Definition.

R416 *derived-type-def* **is** *derived-type-stmt*
[PRIVATE]
component-def-stmt
[*component-def-stmt*]...
end-type-stmt

R417 *derived-type-stmt* **is** [*access-spec*] TYPE *type-name* [(*type-param-name-list*)]

Constraint: A name must not occur more than once in a *type-param-name-list*.

Constraint: If either PRECISION or EXPONENT_RANGE occurs in a *type-param-name-list*, both must occur.

R418 *end-type-stmt* **is** END TYPE [*type-name*]

Constraint: A derived type *type-name* must not be the same as the name of any intrinsic type nor the same as any other accessible derived *type-name*.

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

R419 *component-def-stmt* **is** *type-spec* [[, ARRAY (*explicit-shape-spec-list*)] ::] ■
■ *component-decl-list*

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

Constraint: Each bound in the *explicit-shape-spec* (5.1.2.4.1) must be a nonprecision type-parameter expression (7.1.6.2).

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

R420 *component-decl* **is** *component-name* [(*explicit-shape-spec-list*)] ■
■ [* *char-length*]

Constraint: The * *char-length* option is permitted only if the type specifier is CHARACTER.

Constraint: A *char-length* in a *component-decl* must not contain a *type-param-value* that is an asterisk.

If a component of a derived type is of a type declared to be private, either all components of the derived type must be private or the derived type must be private. A type definition is PRIVATE if the *derived-type-stmt* includes a PRIVATE *access-spec* or if the default accessibility (5.2.3) is PRIVATE and the *derived-type-stmt* does not include a PUBLIC *access-spec*. If a type definition is PRIVATE, the type name, the component names, the type parameter names (if any) and their inquiry functions (13.4.5), and the corresponding structure value constructor (4.4.3) 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 program unit accessing the module via a USE statement. Similarly, the structure constructor for such a type may be employed only within the defining module.

The name of a component of a derived type, provided the name is accessible, may be used to qualify the name of a structured object of this type to select the component of the object. Note that a component may be an array; when selected by component name qualification, such an object is an array even though the parent object may be a scalar object.

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
```

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

```
TYPE LINE
   REAL, ARRAY (2, 2) :: COORD     ! X1, Y1, X2, Y2
   REAL               :: WIDTH     ! LINE WIDTH IN CENTIMETERS
   INTEGER            :: PATTERN   ! 1 FOR SOLID, 2 FOR DASH, 3 FOR DOT
END TYPE LINE

INTEGER, PARAMETER :: SOLID = 1, DASH = 2, DOT = 3

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.

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

```
TYPE POINT
   REAL :: X, Y
END TYPE POINT
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
```

4.4.1.1 Type Parameters of a Derived Type. If a derived-type definition includes a type parameter name list, these names are integer dummy parameters of the definition and may be used as primaries in the expressions for component declarations. When an object of such a type is declared or a value of such a type constructed, actual values for these dummy parameters must be specified. These establish the relevant values for the component attributes via the component type-parameter expressions in which the dummy parameters appear. Any entity, declared within the scoping unit containing the derived-type definition, with the same name as a type parameter is inaccessible within the derived-type definition.

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

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

A type parameter that specifies precision in a derived-type definition must be named PRECISION and is called a **precision type parameter**. PRECISION must be used only to specify

the precision type parameters of real, complex, and derived-type components in the derived-type definition.

A type parameter that specifies exponent range in a derived-type definition must be named EXPONENT_RANGE and is called an **exponent range type parameter**. EXPONENT_RANGE must be used only to specify the exponent range type parameters of real, complex, and derived-type components in the derived-type definition.

If either PRECISION or EXPONENT_RANGE is an actual type parameter for a component, both must be type parameters for the component and both must be dummy type parameters for the derived-type definition containing the component.

A type parameter with a name other than PRECISION or EXPONENT_RANGE is called **nonprecision type parameter**. Neither the precision nor exponent range parameter name may be used as a primary in nonprecision type parameter expressions (7.1.6.2).

A nonprecision type parameter may be used as a primary in any type-parameter expression determining the value for an actual nonprecision type parameter for any component, or determining the bounds of any array component. The dummy type parameters are the only nonconstant primaries permitted in such expressions.

The following example defines a type MATRIX which has all three kinds of type parameters:

```
TYPE MATRIX (PRECISION, EXPONENT_RANGE, ORDER)
   REAL (PRECISION, EXPONENT_RANGE), ARRAY (ORDER, ORDER) :: A
END TYPE MATRIX
```

The following type definition is not valid because it involves prohibited uses of its type parameters:

```
TYPE  NOT_POSSIBLE (PRECISION, EXPONENT_RANGE, ORDER)
   REAL (PRECISION) :: X                ! NO EXPONENT_RANGE PARAMETER.
   REAL (2 * PRECISION) :: DX           ! EXPRESSION INVOLVING PRECISION.
   REAL (ORDER, EXPONENT_RANGE) :: Y    ! NONPRECISION PARAMETER IN A
                                        ! PRECISION CONTEXT.
   REAL (EXPONENT_RANGE, PRECISION), ARRAY (PRECISION, ORDER) :: Z
                                        ! PRECISION PARAMETER IN A NONPRECISION
                                        ! CONTEXT, PRECISION SPECIFIED BY AN
                                        ! EXPONENT_RANGE PARAMETER, AND
                                        ! EXPONENT_RANGE SPECIFIED BY A
                                        ! PRECISION PARAMETER.
END TYPE  NOT_POSSIBLE
```

4.4.1.2 Equivalence of Derived Types. A particular type name may 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 are independent and define different derived types.

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

4.4.2 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.

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

R421 *structure-constructor* **is** *type-name* [(*type-param-spec-list*)] (*expr-list*)

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

The sequence of expressions in a derived-type constructor specifies component values that must agree in number, order, and shape with the components of the derived type. If necessary, each value is converted according to the rules of intrinsic assignment to a value that agrees in type and type parameters with the corresponding component of the derived type. A constructor whose type parameters and values are all constant expressions is a derived-type constant expression.

These examples use the derived types illustrated in 4.4.1 and 4.4.1.1.

```
PERSON (21,'JOHN SMITH')
STRING  (20) (19, 'NOW IS THE TIME FOR')
```

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).

4.4.4 Derived-Type Operations and Assignment. Any operations on derived-type entities and nonintrinsic assignment for derived-type entities must be defined explicitly by operator functions or assignment subroutines. Such definitions are described in Section 12. 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 specified scalar values and interpreted as a rank-one array whose element values are those specified in the sequence. The sequence of values may be specified by any combination of individual scalar values, ranges of values, rank-one arrays, and other array constructors.

R422 *array-constructor* **is** [*array-constructor-value-list*]
or (/ *array-constructor-value-list* /)

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

R423 *array-constructor-value* **is** *scalar-expr*
or *rank-1-expr*
or *scalar-int-expr* : *scalar-int-expr* [: *constructor-stride*]
or [*scalar-int-expr*] *array-constructor*

R424 *rank-1-expr* **is** *expr*

R425 *constructor-stride* **is** *scalar-int-expr*

Constraint: *rank-1-expr* must have rank one.

The second form of *array-constructor-value* is a rank-one expression interpreted as the sequence of scalar values consisting of the elements in array element order (6.2.4.2). The third form of *array-constructor-value* specifies a sequence of values as a rank-one array. The stride, if present, must not be zero. If absent, the default stride is 1.

When the stride is positive, the values specified form a regularly spaced sequence of integers beginning with the first *scalar-int-expr* and proceeding in increments of the stride to the largest such integer not exceeding the second *scalar-int-expr*; the sequence is empty if the first *scalar-int-expr* exceeds the second.

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

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

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

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

```
REAL X (3)
X = [ 3.2, 4.01, 6.5 ]
```

A one-dimensional array may be reshaped into any allowable array shape using the RESHAPE intrinsic function (13.12.77). An example is:

```
Y = RESHAPE (MOLD = [3, 2], SOURCE = [2.0, 2 [4.5], X])
```

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

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

Using the type definitions for PERSON and LINE of 4.4.1, an example of the construction of a derived-type array value is:

```
[PERSON (20, 'SMITH'), PERSON (20, 'JONES')]
```

and an example of the construction of a derived-type scalar value with an array component is:

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

In the latter example, the RESHAPE intrinsic function is used to construct a value which represents a solid line from (0,0) to (1,2) of width 0.1 centimeters.

5 DATA OBJECT DECLARATIONS AND SPECIFICATIONS

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

For example:

```
INTEGER  INCOME, EXPEND
```

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

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

declares three data objects with names X, Y, and Z. These all have default real type and are explicit-shape rank-one arrays with a lower bound of −5, an upper bound of +5, and 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
or REAL [*precision-selector*]
or DOUBLE PRECISION
or COMPLEX [*precision-selector*]
or CHARACTER [*length-selector*]
or LOGICAL
or TYPE (*type-name* [(*type-param-spec-list*)])

R503 *type-param-spec* **is** [*type-param-name* =] *type-param-value*

R504 *type-param-value*
is *specification-expr*
or *

R505 *attr-spec*
is *value-spec*
or *access-spec*
or ALIAS
or ALLOCATABLE
or ARRAY (*array-spec*)
or INTENT (*intent-spec*)
or OPTIONAL
or RANGE [/ *range-list-name* /]
or SAVE

R506 *entity-decl*
is *object-name* [(*array-spec*)] ■
■ [* char-length] [= *constant-expr*]
or *function-name* [* *char-length*]

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

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

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

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

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

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

Constraint: If the ALIAS attribute is specified, no other attribute except the type and ARRAY may be specified.

Constraint: An array declared with an ALIAS or ALLOCATABLE attribute must be specified with an *array-spec* that is a *deferred-shape-spec-list*.

Constraint: The value, accessibility, ALIAS, and SAVE attributes must not be specified for dummy arguments, functions, or objects in a common block. However, the value attribute DATA may be specified using a DATA statement for objects in a named common block, provided the DATA statement appears in a block data program unit.

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

A name that identifies a specific intrinsic function in a scoping unit has a type as specified in 13.11. An explicit type declaration statement is not required; however, it is permitted. If a generic function name appears in a type declaration statement, such an appearance is not sufficient by itself to remove the generic properties from that function.

The *specification-expr* of a nonprecision *type-param-value*, *length-selector*, *char-length*, or *array-spec* may be a nonconstant expression provided the specification expression is in the specification part of a subprogram. 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**. An automatic object must not appear in a SAVE or DATA statement nor be declared with a SAVE or DATA attribute.

Examples of type declaration statements are:

```
LOGICAL, ARRAY (5, 5) :: MASK1, MASK2
REAL (PRECISION = 10) A (10)
TYPE (STRING (10)) :: S  ! TYPE STRING IS DEFINED IN 4.4.1.1
COMPLEX, DATA :: CUBE_ROOT = (-0.5, 0.866)
```

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

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

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

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

Constraint: The *type-param-value* for a precision type parameter must be either a specification expression in which no primary is a reference to a variable except as the argument of the EFFECTIVE_PRECISION function or an asterisk. The *type-param-value* for an exponent range type parameter must be either a

specification expression in which no primary is a reference to a variable except as the argument of the EFFECTIVE_EXPONENT_RANGE function or an asterisk.

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

If either *p* or *r* is an asterisk, both must be asterisks and all entities being declared in the statement must be dummy arguments. The asterisks specify that the type parameter values will be assumed from the corresponding actual arguments. In a procedure reference, all actual arguments associated with dummy arguments having precision and exponent range parameters specified as asterisks must have the same declared precision value and the same declared exponent range value. If all dummy arguments having precision and exponent range parameters specified as asterisks are optional, at least one must be present in each reference to the procedure.

If either part of the precision selector is omitted, a processor-dependent default value is used for the omitted type parameter, which is regarded as different from any explicitly specified value in any context that requires type parameters to agree.

If the precision selector is omitted entirely, a processor-dependent default approximation method is selected and the entities declared are of the default real type. Their type parameter values are regarded as different from those of double precision real and any that are explicitly specified in any context that requires type parameters to agree.

5.1.1.3. DOUBLE PRECISION. The DOUBLE PRECISION type specifier specifies that entities whose names are declared in this statement are of intrinsic type double precision real (4.3.1.2). Their type parameter values are regarded as different from those of default real and any that are explicitly specified in any context that requires type parameters to agree.

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

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

If either the precision or exponent range parameter of an object of type complex is specified as an asterisk, both must be asterisks and all entities being declared in the statement must be dummy arguments. The asterisks specify that the type parameter values will be assumed from the corresponding actual arguments. In a procedure reference, each actual argument associated with a dummy argument having precision or exponent range parameters specified as asterisks must have the same declared precision value and the same declared exponent range value (5.1.1.2).

If all dummy arguments having precision and exponent range parameters specified as asterisks are optional, at least one must be present in each reference to the procedure.

If either part of the precision selector is omitted, a processor-dependent default real value is used for the omitted type parameter, which is regarded as different from any explicitly specified value in any context that requires type parameters to agree.

If the precision selector is omitted entirely, a processor-dependent approximation method is selected and the objects declared are of the default complex type. In any context that requires type parameters to agree, their type parameter values are regarded as different from any that are explicitly specified, except that they are the same as for the default real type.

Examples of type declarations with precision selectors are:

```
REAL (10, 15) A
REAL (PRECISION = 10) B
COMPLEX (EXPONENT_RANGE = 60) :: C
```

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

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

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

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

If the type parameter value evaluates to a negative value, the length of character entities declared is zero. A type parameter value of * may be used only in the following ways:

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

(2) A type parameter value of * 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 itself may be specified with a type parameter value of *; in this case, any scoping unit invoking the function must declare this function name with a type parameter value other than * or access such a definition. When the function is invoked, the length of the result variable in the function is assumed from the value of this type parameter.

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

Examples of character type declaration statements are:

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

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

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

Each type parameter value is associated with the corresponding type parameter name in a manner similar to the association of arguments in a procedure reference (12.4). The association may be positional or the type parameter names may be used as keywords, as with procedure arguments.

Objects declared with a type parameter value specified as an asterisk must be dummy arguments in a procedure. The value for such a parameter is assumed from the actual argument that becomes associated with the dummy argument when the procedure is referenced. If the derived-type definition defining the type has precision and exponent range parameters and either the precision or exponent range parameter is specified as an asterisk, both must be asterisks. In this case, only one value for precision and one value for exponent range is assumed on any reference to the procedure. Hence, all dummy arguments that have a

precision and exponent range specified as asterisks assume the same value for precision and the same value for exponent range. All actual arguments that become associated with such dummy arguments must be declared with the same value for precision and the same value for exponent range. For a type parameter other than precision or exponent range, the value specified as an asterisk is assumed independently for each object.

The above rules may be demonstrated as follows. Let a derived type with precision and exponent range parameters be defined in a module as:

```
MODULE TYPE_DEFINER
   TYPE NODE (PRECISION, EXPONENT_RANGE, M)
     REAL (PRECISION, EXPONENT_RANGE) :: DOT
     CHARACTER (M)                    :: DASH
   END TYPE NODE
END MODULE TYPE_DEFINER
```

Suppose there exists a calling routine and a called procedure. The following code segment declares the actual arguments in the calling routine:

```
USE TYPE_DEFINER
TYPE (NODE (10, 50, 2)) :: ALPHA
REAL (10, 50)           :: BETA
CALL SUB (ALPHA, BETA)
```

The following code segment demonstrates permissible declarations for the dummy arguments in the called routine:

```
SUBROUTINE SUB (ALPHA_ARG, BETA_ARG)
USE TYPE_DEFINER
TYPE (NODE(*, *, *)) :: ALPHA_ARG
REAL (*, *)          :: BETA_ARG
```

Now suppose the caller had declared ALPHA and BETA as follows:

```
TYPE (NODE (10, 50, 2)) :: ALPHA
REAL                    :: BETA
```

where the values for the precision and exponent range of the default real type are also 10 and 50, respectively. The declarations in the subroutine code segment shown above would not be permitted because both dummy arguments must assume the same value for precision and the same value for exponent range. The declarations in the caller are considered to be different (even though the values happen to be the same) because one is a specific declaration of precision and exponent range while the other is a default declaration.

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

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

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

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

Examples of declarations with a value attribute are:

```
REAL, PARAMETER :: ONE = 1.0, Y = 4.1 / 3.0
INTEGER, DATA :: NEXT = 1
INTEGER, ARRAY (3), PARAMETER :: ORDER = [1, 2, 3]
```

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

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

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

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

The presence of a DATA attribute implies that all the data objects declared in this statement are saved, except for data objects in a named common block. The implied SAVE attribute may be reaffirmed by explicit use of the SAVE attribute in a type declaration statement, or by the inclusion of the object names in a SAVE statement (5.2.4). The DATA attribute must not be specified for 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 alias object, an allocatable array, or an automatic object.

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

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

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. The default for entities without an explicitly specified *access-spec* is PUBLIC, but this may be changed by a PRIVATE statement (5.2.3).

An example of an accessibility specification is:

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

5.1.2.3. INTENT Attribute. The **INTENT attribute** may be specified only for a dummy argument that is not allocatable and may appear in the *declaration-construct* of a subprogram or interface block (12.3.2.1). An INTENT attribute specifies the intended use of the dummy argument.

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

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

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

The INTENT (INOUT) attribute specifies that the dummy 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 must be definable.

If no INTENT attribute is specified for a dummy argument, its use is subject to the limitations of the associated actual argument (12.5.2.1, 12.5.2.2, 12.5.2.3).

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

An example of an INTENT specification is:

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

5.1.2.4. ARRAY Attribute. The **ARRAY attribute** specifies that entities whose names are declared in this statement are arrays. The rank and shape are specified by the *array-spec* in the *entity-decl* if there is one, or by the *array-spec* in the ARRAY attribute, 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 ARRAY attribute. If the ARRAY attribute is omitted, an *array-spec* must be specified in the *entity-decl* to declare an array.

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.

A *deferred-shape-spec* is used to declare both an allocatable array and an alias array.

Examples of array attribute specifications are:

```
SUBROUTINE EX (N, A, B, S)
   REAL, ARRAY (N, 10) :: W           ! EXPLICIT-SHAPE ARRAY
   REAL A (:), B (0:)                 ! ASSUMED-SHAPE ARRAYS
   REAL, ALIAS :: D (:, :)            ! DEFERRED-SHAPE ARRAY
   REAL, ALLOCATABLE, ARRAY (:) :: E ! DEFERRED-SHAPE ARRAY
   REAL :: S (N, *)                   ! ASSUMED-SIZE 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 dimension bounds of the array.

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

R515 *lower-bound* **is** *scalar-int-expr*

R516 *upper-bound* **is** *scalar-int-expr*

Constraint: An explicit-shape array whose bounds depend on the values of nonconstant expressions must be a dummy argument, a function result, or a local array of a procedure.

Constraint: The bounds in an explicit-shape array declaration must be specification expressions (7.1.6.3).

If an explicit-shape array is an automatic array or a dummy argument that has bounds that are nonconstant specification expressions, the bounds, and hence shape, 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.

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 declared subscript range of the array in that dimension is the set of integer values between and including the lower and upper bounds, provided the upper bound is not less than the lower bound. If the upper bound is less than the lower bound, the range is empty, the extent in that dimension is zero, and the array is of zero size. If the *lower-bound* is omitted, the default value is 1. The number of sets of bounds specified is the rank.

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

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

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

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

The size of a dimension of an assumed-shape array is the size of the corresponding dimension of the associated actual argument array. If the lower bound value is d and the size of the corresponding dimension of the associated actual argument array is s, then the value of the declared upper bound is $s + d - 1$. If the lower bound is omitted, the default value is 1. The declared 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 alias array. An **allocatable array** is a named array whose type, type parameters, name, and rank are specified in a type declaration statement containing an ALLOCATABLE attribute, but whose declared bounds, and hence declared shape, are determined when space is allocated for the array by execution of an ALLOCATE statement (6.2.2).

An **alias array** is a named array whose type, type parameters, name, and rank are specified in a type declaration statement containing an ALIAS attribute, but whose bounds, and hence shape, are declared when it is alias associated to an array by execution of an IDENTIFY statement (6.2.6).

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

The bounds of the allocated array are unaffected by any subsequent redefinition or undefinition of specification expression variables involved in the bounds.

The size, bounds, and shape of an alias array that is not alias associated are undefined. No reference may be made to any part of it except by using certain intrinsic functions, nor may any part of it be defined. The declared lower and upper bounds of each dimension are those specified in the IDENTIFY statement when the array is alias associated.

An allocatable dummy array argument may be associated only with an allocatable actual argument. An actual argument that is an allocated array may be associated with a nonallocatable array dummy argument. An array-valued function may declare its result to be an allocatable array.

5.1.2.4.4 Assumed-Size Array. An **assumed-size array** is a dummy 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: An *assumed-size-spec* must not appear in an ARRAY attribute.

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

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

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

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

(3) If the actual argument is a character array, character array element, or a 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-specs*.

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

An assumed-size array has no upper bound in its last dimension and therefore has no shape or size other than the size assumed from an associated actual argument.

The declared and effective bounds of the first $n - 1$ dimensions are those specified by the *explicit-shape-spec-list*, if present, in the *assumed-size-spec*. The declared and effective lower bound of the last dimension is *lower-bound*, if present, and one otherwise. An assumed-size array may be subscripted or sectioned (6.2.4). If a section subscript list is provided in which a subscript or section with an upper bound for the last dimension is present, the resulting array has a declared shape and size.

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

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

Objects in the scoping unit of a module may be declared with a SAVE attribute. Such objects retain their definition status, allocation status, effective range, and value when any procedure that accesses the module in a USE statement executes a RETURN or END statement. The SAVE attribute must 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 alias.

5.1.2.6 OPTIONAL Attribute. The **OPTIONAL attribute** may be specified only in the scoping unit of a subprogram or an interface block, and may 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 (12.5.2.8).

5.1.2.7 ALIAS Attribute. The **ALIAS attribute** specifies that only the type, type parameters, rank, and name of the objects declared in the statement are specified. The object must not be referenced or defined unless, as a result of executing an IDENTIFY statement (6.2.6), it is alias associated with an object that may be referenced or defined. If it is an array, it must

be declared with an *array-spec* that is a *deferred-shape-spec-list*. The array does not have a shape unless it is alias associated. Examples of ALIAS attribute specifications are:

```
REAL, ARRAY (:, :), ALIAS :: B
REAL, ALIAS :: C
```

5.1.2.8 RANGE Attribute. The **RANGE attribute** may be specified only for a nonassumed-size array object that is not a function result and specifies that the array may have an effective shape that differs from its declared shape (6.2.1.2). The appearance of an array name in an executable construct specifies the set of elements determined by the effective shape. This set of elements is known as the **range**. The range is established or changed by execution of a SET RANGE statement. The effective shape of a rangeable array is undefined prior to it being established by the execution of a SET RANGE statement. For an alias or allocatable array, the effective shape and declared shape following the execution of an IDENTIFY or ALLOCATE statement are the same and equal to the shape given in the IDENTIFY or ALLOCATE statement.

If the range list name is omitted, the arrays declared in that type declaration may have different shapes, and the individual array names may appear explicitly in SET RANGE statements. If the range list name is specified, the arrays must all be saved or all not be saved and the arrays must all have explicit shapes, must be declared with the same rank, lower bounds, and upper bounds, and may be reshaped only by execution of a SET RANGE statement containing that range list name.

If a range list has the PUBLIC attribute, no array in the list may have the PRIVATE attribute.

5.2 Attribute Specification Statements.

Most of the attributes (other than type) may be specified for entities, independently of type, by single attribute specification statements. An attribute declared by an attribute specification statement has exactly the same restrictions and properties it would have if declared as an attribute specifier in a type declaration statement. An entity must not be explicitly given any of the following attributes more than once in a scoping unit: type, value, accessibility, intent, array, save, optional, alias, and range.

5.2.1 INTENT Statement.

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

Constraint: An *intent-stmt* may occur only in the scoping unit of a subprogram or an interface block.

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

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* may occur only in the scoping unit of a subprogram or an interface block.

This statement specifies that any of the specified dummy arguments need not be associated with an actual argument on an invocation of the procedure (12.5.2.8). Each specified argument has the OPTIONAL attribute.

An example of an OPTIONAL statement is:

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

5.2.3 Accessibility Statements.

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

Constraint: An *access-stmt* may appear only in the scoping unit of a module or of a derived-type definition contained in a module. If it appears in a derived-type definition, it must be a PRIVATE statement and must not have a *use-name-list*. Only one accessibility statement with an omitted *use-name-list* is permitted in the scoping unit of a module; however, more than one PRIVATE statement may appear if each one is contained in a different scoping unit of either a derived-type definition or a module.

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

This statement declares the accessibility, PUBLIC or PRIVATE, of the entities (5.1.2.2).

If the use-name-list is omitted, the statement sets the default accessibility that applies to all potentially accessible entities in the scoping unit of the module. The statement

```
PUBLIC
```

confirms the default of public accessibility. The statement

```
PRIVATE
```

sets the default to private accessibility.

Examples of accessibility statements are:

```
MODULE EX
   PRIVATE
   PUBLIC :: A, B, C
```

5.2.4 SAVE Statement.

R523 *save-stmt* **is** SAVE [[::] *saved-object-list*]

R524 *saved-object* **is** *object-name*
or / *common-block-name* /

Constraint: An *object-name* must not be a dummy argument name, a procedure name, a function result name, an automatic data object, an alias name, a range list name, a namelist group name, or the name of an object in a common block.

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

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

A SAVE statement with an empty saved object list is treated as though it contained the names of all allowed items.

An example of a SAVE statement is:

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

5.2.5 DIMENSION Statement.

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

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

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

An example of a DIMENSION statement is:

```
DIMENSION A (10), B (10, 70), C (-3:12, *)
```

5.2.6 DATA Statement. A **DATA statement** is used to provide initial values for variables. There are two forms of the DATA statement. The first form in R526 is the **list-oriented DATA statement** and the second form is the **object-oriented DATA statement**.

R526 *data-stmt* **is** DATA *data-stmt-set* [[,] *data-stmt-set*]...
or DATA (*data-value-def-list*)

A variable, or part of a variable, must not be initialized more than once in an executable program.

A variable that appears in a DATA statement and is typed implicitly may appear in a subsequent type declaration only if that declaration confirms the implicit typing.

If a named variable or part of a named variable is initialized in a DATA statement, the named variable has the SAVE attribute, and this may be reaffirmed by a SAVE statement or a type declaration statement containing the SAVE attribute.

The declared range (6.2.1.2) of an array is used in determining the array elements of an unqualified array name or an array section in a DATA statement.

5.2.6.1 List-Oriented DATA Statement.

R527 *data-stmt-set* **is** *data-stmt-object-list* / *data-stmt-value-list* /

R528 *data-stmt-object* **is** *object-name*
or *array-element*
or *substring*
or *data-implied-do*

Constraint: The *parent-string* in *substring* (R605) must not be a *scalar-constant*.

R529 *data-stmt-value* **is** [*data-stmt-repeat* *] *data-stmt-constant*

R530 *data-stmt-constant* **is** *constant*
or *signed-int-literal-constant*
or *signed-real-literal-constant*

R531 *data-stmt-repeat* **is** *scalar-int-constant*

R532 *data-implied-do* **is** (*data-i-do-object-list*, *data-i-do-variable* = ■
■ *scalar-int-expr*, *scalar-int-expr* [, *scalar-int-expr*])

R533 *data-i-do-object* **is** *array-element*
or *substring*
or *data-implied-do*

R534 *data-i-do-variable* **is** *scalar-int-variable*

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

Constraint: The data statement repeat factor must be positive. If the data statement repeat factor is a named constant, it must have been declared previously in the scoping unit or made accessible by use or host association.

Constraint: A variable whose name is included in a *data-stmt-object-list* or a *data-i-do-object-list* must not be of a derived type, a structure component, a dummy argument, made accessible by use or host association, in a named common block unless the DATA statement is in a block data program unit, in a blank common block, an alias object, a character string with zero length, or a function name. An object whose name is included in either of the above object lists must not be an automatic object, a deferred-shape array, or a zero-sized array.

Constraint: The only variables that may appear in subscripts of the *array-element* in a *data-i-do-object* (R533) are DO variables from the same level or an outer level of the *data-implied-do*. Each such DO variable must appear in some subscript of the *array-element*.

The *data-stmt-object-list* is expanded to form a sequence of scalar variables. An array whose unqualified name appears in a *data-stmt-object-list* is equivalent to a complete sequence of its elements in array element order (6.2.4.2). A *data-implied-do* is expanded to form a sequence of array elements, under the control of the implied-do DO variable, as in the DO loop (8.1.4.1, 9.4.2). A subscript in an array element *data-i-do-object* must be an expression whose primaries are either constants or DO variables of the containing *data-implied-dos*. A *scalar-int-expr* of a *data-implied-do* must involve as primaries only constants or DO variables of the containing *data-implied-dos*.

The *data-stmt-value-list* is expanded to form a sequence of constant values. Each value must be a constant that is either previously defined or made accessible by a use or host association. A data statement repeat factor indicates the number of times the following constant is to be included in the sequence; omission of a data statement repeat factor has the effect of a repeat factor of one.

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

If an object is of type character, derived type, or logical, the corresponding constant must be of the same type. When the object is of type integer, real, or complex, the corresponding constant must also be of type integer, real, or complex. Note that if an object is of type double precision real and the constant is of type real, the processor may supply more precision derived from the constant than can be contained in a real datum with the type parameters of the constant.

The value of the constant must be assignment compatible with its corresponding variable, according to the rules of intrinsic assignment (7.5.1.4), and the constant defines the initial value of the variable according to those rules.

Examples of list-oriented DATA statements are:

```
CHARACTER (LEN = 10)  NAME
INTEGER, ARRAY (0:9) :: MILES
REAL, ARRAY (100, 100) :: SKEW
DATA  NAME / 'JOHN DOE' /, MILES / 10*0 /
DATA  ((SKEW (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 /
```

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.

5.2.6.2 Object-Oriented DATA statement.

R535 *data-value-def* **is** *variable* = *constant-expr*
or *data-init-implied-do* = *data-init-implied-do-value*

R536 *data-init-implied-do* **is** (*data-init-implied-do-object* , *data-init-implied-do-control*)

R537 *data-init-implied-do-object* **is** *array-element*
or *data-init-implied-do*

R538 *data-init-implied-do-control* **is** *data-i-do-variable* = ■
■ *scalar-int-expr* , *scalar-int-expr* [, *scalar-int-expr*]

R539 *data-init-implied-do-value* **is** *array-constructor*

Constraint: Neither the name of *variable* in *data-value-def* (R535) nor the name of *array-element* in *data-init-implied-do-object* (R537) can be accessible names of the whole or part of dummy arguments, procedures, function results, automatic objects, allocatable arrays, alias objects, or objects in a common block.

Constraint: The only variables that may appear in subscripts of the *array-element* in a *data-init-implied-do-object* (R537) are DO variables from the same level or an outer level of the *data-init-implied-do*. Each such DO variable must appear in some subscript of the *array-element*.

Constraint: Each *data-implied-do* control portion or each *data-init-implied-do-control* must conform to the rules of the DO construct (8.1.4.1), except that the DO variable must be of type integer. The only variables that may appear in *scalar-int-expr* are DO variables from an outer *data-init-implied-do-control.*

Constraint: Each element of the array constructor must be a scalar constant expression.

The size of the *array-constructor* must be equal to the number of elements referenced by the *data-init-implied-do-control*s.

The *data-init-implied-do* assignment is performed as if:

(1) The set of *data-init-implied-do-control*s are converted to nested DO constructs with the outermost control being the outermost construct, and the innermost control being the innermost construct and with the array element assignment appearing inside the innermost construct.

(2) The assignments are made from the *array-constructor* in array element order in accordance with the rules of intrinsic assignment (7.5.1.4).

The *variable* in *data-value-def* (R535) becomes defined with the value determined from the *constant-expr* that appears on the right of the equal sign in accordance with the rules of intrinsic assignment.

Examples of object-oriented DATA statements are:

```
REAL S
REAL, ARRAY (1:10) :: A
REAL, ARRAY (10, 10) :: B
INTEGER I, J, K, L, M, N
DATA (I = 1, J = 1, S = 0.0)
DATA ((A (K), K = 1, 9, 2) = [5 [1.0]])
DATA (((B (M, N), M = 1, N), N = 1, 10) = [55 [0.0]], L = 10)
```

The odd elements of A are initialized to 1.0. The upper triangle of B is initialized to zero.

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

R540 *parameter-stmt* **is** PARAMETER (*named-constant-def-list*)

R541 *named-constant-def* **is** *named-constant* = *constant-expr*

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

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

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

Each named constant has the PARAMETER attribute.

An example of a PARAMETER statement is:

```
PARAMETER (ONE = 1.0, SQRT2 = SQRT (2.0))
```

5.2.8 RANGE Statement. A **RANGE statement** specifies the RANGE attribute (5.1.2.8) for each array name in the array name list.

R542 *range-stmt* **is** RANGE [/ *range-list-name* /] *array-name-list*

Constraint: An array name in *array-name-list* must not be the name of a function procedure.

If the range list name is present, the arrays in the array name list must all be explicit-shaped arrays declared with the same rank, lower bounds, and upper bounds, but they may be of any type. The arrays must all be saved or all not be saved. The effective shape of all arrays in the array name list may be changed by the execution of a SET RANGE statement containing only the range list name.

The *array-name-list* following each successive appearance of the same *range-list-name* in the same scoping unit is treated as a continuation of the list for that *range-list-name*.

If the range list name is omitted, the arrays in the array name list may each be an explicit-shaped array, assumed-shape array, allocatable array, or alias array and they may have different ranks, lower bounds, and upper bounds. Each array name may appear independently in different SET RANGE statements.

Examples of RANGE statements are:

```
RANGE /RLIST/ A, B, C
RANGE X, Y
```

5.3 IMPLICIT Statement. An **IMPLICIT statement** specifies a type, and possibly type parameters, for all implicitly typed data objects 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.

R543 *implicit-stmt* **is** IMPLICIT *implicit-spec-list*
or IMPLICIT NONE

R544 *implicit-spec* **is** *type-spec* (*letter-spec-list*)

R545 *letter-spec* **is** *letter* [– *letter*]

Constraint: If the minus and second letter appear, the second letter must 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 ot A, B, C. If IMPLICIT NONE is specified, there must be no other IMPLICIT statements in the scoping unit. The same letter must not appear as a single letter, or be included in a range of letters, more than once in all of the IMPLICIT statements in a 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 is the mapping in the host scoping unit. A program unit is treated as if it had a host with the declaration

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

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 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 data entity is treated as if it were declared in an explicit type declaration in the outermost scoping unit in which it appears.

The following are examples of the use of IMPLICIT statements:

```
MODULE MOD
   IMPLICIT NONE
   ...
   INTERFACE
   FUNCTION FUN (I)    ! All data entities must
   INTEGER FUN, I      ! be declared explicitly
   END INTERFACE
CONTAINS
   FUNCTION JFUN (J)   ! All data entities must
   INTEGER JFUN, J     ! be declared explicitly.
   ...
   END FUNCTION JFUN
END MODULE MOD

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
```

```
   SUBROUTINE SUB3
      USE MOD          ! Accesses the 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
```

An IMPLICIT statement may specify a *type-spec* of derived type. For example, given a type defined as follows:

```
TYPE POSN (PRECISION, EXPONENT_RANGE)
   REAL (PRECISION, EXPONENT_RANGE) 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, precision 10 and exponent range 32, and the remaining variables are implicitly typed with type INTEGER by using the following IMPLICIT statement:

```
IMPLICIT TYPE (POSN (10, 32)) (A-B, W-Z), INTEGER (C-V)
```

5.4 NAMELIST Statement.

A **NAMELIST statement** specifies a group of named data objects which can then be referred to by a single name for the purpose of data transfer (9.4, 10.9).

R546 *namelist-stmt* **is** NAMELIST / *namelist-group-name* / *namelist-group-object-list* ■
■ [[,] / *namelist-group-name* / *namelist-group-object-list*]...

R547 *namelist-group-object* **is** *variable-name*

Constraint: A *namelist-group-object* must not be an array dummy argument with nonconstant bounds, a variable with assumed parameters, an automatic object, an alias object, or a deferred-shape array.

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

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* 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 either must have its type, type parameters, and shape specified by a previous occurrence in a type declaration statement in the same scoping unit, or must be determined by the implicit typing rules currently 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 must confirm this implied type.

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 and the COMMON statement provide for control of the "order" and "layout" of storage units. Section 14.7.2 describes the general mechanism of storage association.

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

R548 *equivalence-stmt* **is** EQUIVALENCE *equivalence-set-list*

R549 *equivalence-set* **is** (*equivalence-object* , *equivalence-object-list*)

R550 *equivalence-object* **is** *variable-name*
or *array-element*
or *substring*

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

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

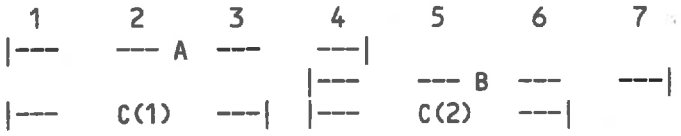
5.5.1.1 Equivalence Association. An EQUIVALENCE statement specifies that the storage sequences of the data objects whose names appear in an *equivalence-set* have the same first storage unit. This causes the storage association of the data objects in the *equivalence-set* and may cause storage association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

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

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

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

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



5.5.1.3 Array Names and Array Element Designators. If an array element designator appears in an EQUIVALENCE statement, the number of subscripts must be the same as the rank of the array.

The use of an 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 must not specify that the same storage unit is to occur more than once in a storage sequence. For example,

```
REAL, ARRAY (2) :: A
REAL :: B
EQUIVALENCE (A (1), B), (A (2), B) ! NOT STANDARD CONFORMING
```

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

```
REAL A (2)
DOUBLE PRECISION D (2)
EQUIVALENCE (A (1), D (1)), (A (2), D (2)) ! 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 an executable program. Thus, the COMMON statement provides a global data facility based on storage association (14.7.2). The common blocks specified by the COMMON statement may be named and are called **named common blocks**, or may be unnamed and are called **blank common**.

R551 *common-stmt* **is** COMMON [/ [*common-block-name*] /]
■ *common-block-object-list* ■
■ [[,] / [*common-block-name*] / ■
■ *common-block-object-list*]...

R552 *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-lists* within a scoping unit.

Constraint: A *common-block-object* must not be a dummy argument, an object of derived type, an alias object, an allocatable array, an automatic object, an object of real type unless of default real type or double precision real type, an object of complex type unless of default complex type, an array of zero size, a character string of zero length, or a function name.

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

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 list are specified to be in blank common. Alternatively, the appearance of two slashes with no common block name between them declares the data objects whose names appear in the common block list that follows to be in blank common.

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

If an object of type character is in a common block, all of the entities in that common block must be of type character.

An array in a common block may have the RANGE attribute.

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

(1) A storage sequence is formed consisting of the storage sequences of all data objects in the common block object lists for the common block. The order of the storage sequence 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.

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

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 named common blocks to become undefined unless the common block name has been declared in a SAVE statement, but never causes data objects in blank common to become undefined (14.8.6).

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

(3) A data object in a named common block may be initially defined by means of a DATA statement in a block data program unit, but objects in blank common must not be initially defined (11.4).

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

```
COMMON /X/ A
REAL B (2)
EQUIVALENCE (A, B (2))     ! NOT STANDARD CONFORMING
```

A common block may be declared in a module (11.3). If it is, it must not be declared in another scoping unit that accesses entities from the module. The name of a PUBLIC data object accessible from a module must not appear in a COMMON or EQUIVALENCE statement in the scoping unit containing the USE statement or in scoping units contained within the scoping unit containing the USE statement.

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 data object becomes defined with a value when the data object name or subobject designator appears in certain contexts and when certain events occur (14.8).

A data object that is not a constant is a **variable**.

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

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

R602 *logical-variable* **is** *variable*

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

R603 *char-variable* **is** *variable*

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

R604 *int-variable* **is** *variable*

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

Under some circumstances, alias variables (5.1.2.7), allocatable arrays (5.1.2.4.3), dummy arguments, and variables associated with dummy arguments (12.5.2.1, 12.5.2.8) must not be defined.

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 (5.1.2.1.1, 5.2.7). A reference to a constant is always permitted; redefinition of a constant is never permitted.

For example, given the declarations:

```
CHARACTER (10)  A, B (10)
TYPE (PERSON)  P    ! SEE 4.4.1
```

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

6.1. Scalars. A **scalar** (2.4.6) is a data entity that is not array valued. Its value, if defined, is a single element from the set of values that characterize its data type.

A scalar has rank zero.

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

R605 *substring* **is** *parent-string* (*substring-range*)

R606 *parent-string* **is** *scalar-variable-name*
or *array-element*
or *scalar-structure-component*
or *scalar-constant*

R607 *substring-range* **is** [*scalar-int-expr*] : [*scalar-int-expr*]

Constraint: *parent-string* must 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 (*ending-point* − *starting-point* + 1, 0).

Let the characters in the parent string be numbered 1, 2, 3, ..., *n*, where *n* is the length of the parent string. Then the characters in the substring are those from the parent string from the starting point and proceeding in sequence up to and including the ending point. Both the starting point and the ending point must be within the range 1, 2, ..., *n* unless the starting point exceeds the ending point, in which case the substring has length zero. 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. If the parent is an array section (6.2.4.3), the substring is an array of the same shape as the array section and each element is the designated substring of the corresponding element of the array section.

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

R608 *structure-component* **is** *parent-structure* % *component-name*

R609 *parent-structure* **is** *scalar-variable-name*
or *array-variable-name*
or *array-element*
or *array-section*
or *structure-component*
or *named-constant*

Constraint: *parent-structure* must be of derived type.

The type of the structure component is the same as the type declared for the component in the derived-type definition. Each type parameter, if any, of a structure component is declared for the component in the derived-type definition (4.4.1) and is either a constant or is a type parameter of a derived type (4.4.1.1) whose actual value is established in the declaration of a parent object or component (5.1.1.7, 4.4.1).

The resulting data subobject is an array if either the parent structure is an array or the component is an array, but not both.

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
```

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, except where array element ordering (6.2.4.2) is specified.

6.2.1 Whole Arrays. A **whole array** is a named array.

6.2.1.1 Array Constants and Variables. A whole array is either a named constant or variable. A **whole array named constant** is the name of a constant expression (5.1.2.1.1 and 5.2.7) and comprises those elements determined by the declared shape of the named constant. A **whole array variable** is the name of a variable that is an array; the name does not have a subscript list appended to it.

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

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

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

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

R610 *allocate-stmt* **is** ALLOCATE (*array-allocation-list* ■
■ [, STAT = *stat-variable*])

R611 *stat-variable* **is** *scalar-int-variable*

Constraint: The *stat-variable* must not be allocated within the ALLOCATE statement in which it appears.

R612 *array-allocation* **is** *array-name* (*explicit-shape-spec-list*)

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

Constraint: A bound in an *array-allocation explicit-shape-spec* is not restricted to a specification expression, but must not be an expression involving as a primary an array inquiry function (13.9.11) whose argument is any other array in the same ALLOCATE statement.

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

An example of an ALLOCATE statement is:

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

At the time the ALLOCATE statement is executed, the values of the lower bound and upper bound expressions in an explicit shape specification (5.1.2.4.1) determine the declared bounds of an allocatable array. Subsequent redefinition or undefinition of any entities in the bound expressions do not affect the array shape.

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.

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

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

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

R613 *deallocate-stmt* **is** DEALLOCATE (*array-name-list* ■
■ [, STAT = *stat-variable*])

Constraint: The *stat-variable* must not be deallocated within the same DEALLOCATE statement in which it appears.

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.

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

Deallocating an array that is not currently allocated causes an error condition in the DEALLOCATE statement. When the execution of a procedure is terminated by execution of a RETURN or END statement, any array allocated within the procedure is deallocated unless it is one of the following:

(1) An allocatable dummy argument or function result,

(2) An allocatable array with the SAVE attribute,

(3) An allocatable array in the scoping unit of a module if the module also is accessed by another scoping unit that is currently in execution.

Such allocated arrays retain their definition status at the execution of the RETURN or END statement.

An example of a DEALLOCATE statement is:

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

6.2.4. Array Elements and Array Sections.

R614 *array-element* **is** *parent-array* (*subscript-list*)

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

R615 *array-section* **is** *parent-array* (*section-subscript-list*) [(*substring-range*)]

Constraint: If *substring-range* is present, *parent-array* must be of type character.

Constraint: At least one *section-subscript* must be a *subscript-triplet*.

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

R616 *parent-array* **is** *array-name*
or *structure-component*

Constraint: A *structure-component* may appear only if the component specified is an array.

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

R618 *section-subscript* **is** *subscript*

or *subscript-triplet*

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

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

An array element is a scalar. An array section is an array. If a *substring-range* is present in an *array-section*, the object is an array of the shape specified by the *section-subscript-list* and each element is the designated substring of the corresponding element of the array section. 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.

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

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

Table 6.1. Subscript Order Value

Rank	Explicit Shape Specifier	Subscript List	Subscript Order Value
1	$j_1{:}k_1$	s_1	$1+(s_1-j_1)$
2	$j_1{:}k_1, j_2{:}k_2$	s_1, s_2	$1+(s_1-j_1)$ $+(s_2-j_2)\times d_1$
3	$j_1{:}k_1, j_2{:}k_2, j_3{:}k_3$	s_1, s_2, s_3	$1+(s_1-j_1)$ $+(s_2-j_2)\times d_1$ $+(s_3-j_3)\times d_2\times d_1$
.	.	.	.
.	.	.	.
.	.	.	.
7	$j_1{:}k_1, \ldots, j_7{:}k_7$	$s_1, \ldots, s_7$	$1+(s_1-j_1)$ $+(s_2-j_2)\times d_1$ $+(s_3-j_3)\times d_2\times d_1$ $+\cdots$ $+(s_7-j_7)\times d_6$ $\times d_5\times\cdots\times d_1$

Notes for Table 6.1:

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

(2) If the size of the array is nonzero, $j_i \le s_i \le k_i$ for all $i = 1, 2, \ldots, 7$.

6.2.4.3 Array Sections. An **array section** is an array subobject designated by an array name or designator with a section subscript list, optionally followed by a substring range. Each subscript in a section subscript must be within the declared bounds for that dimension, unless the subscript triplet defines an empty sequence.

Each subscript triplet in the section subscript list indicates a sequence of subscripts (6.2.4.4). 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.

The rank of the array section is the number of subscript triplets 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. 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.

6.2.4.4 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 effective lower bound for the array and an omitted second subscript is equivalent to the effective upper bound (6.2.5). An omitted stride is equivalent to a stride of one.

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

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

The stride must not be zero.

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

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

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

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

6.2.5 SET RANGE Statement. Execution of a **SET RANGE statement** establishes the effective ranges for the arrays in the array name list or for the members of the range list specified by the range list name (5.2.8).

R621 *set-range-stmt* **is** SET RANGE ([*effective-range-list*]) *array-name-list*
or SET RANGE ([*effective-range-list*]) / *range-list-name* /

R622 *effective-range* **is** *explicit-shape-spec*
or [*lower-bound*] : [*upper-bound*]

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

Constraint: All arrays being ranged must have the same rank and declared lower and upper bounds in corresponding dimensions.

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

The entire effective range list is evaluated before any effective ranges are changed. Each effective range specifies the effective lower and upper bounds for the corresponding dimension of each array in *array-name-list* or range list. The bounds are not restricted to specification expressions. Once the effective range has been set by execution of a SET RANGE statement, subsequent redefinition or undefinition of entities used in the bounds expressions have no effect on the effective ranges.

An array name must not appear in the array name list of a SET RANGE statement unless it has the RANGE attribute. A SET RANGE statement must not be used to establish the effective range for an allocatable array that is not allocated or an alias array that is not alias associated. The values of each effective lower bound and each effective upper bound must be within the declared bounds for the corresponding dimension of every array in the array list or every member of the range list specified by the range list name unless the effective size of that particular dimension is zero. The effect of a SET RANGE is global to all scoping units accessing those arrays by use association. If a lower bound or an upper bound of an effective range is omitted, the default value is the current effective lower bound or effective upper bound, respectively, for each array being ranged. If the effective range list is omitted, the effective lower bounds and the effective upper bounds are set to the declared lower and upper bounds, respectively, for each array being ranged.

A rangeable array whose effective shape is undefined must not be referenced or defined except via an array element or an array section reference that does not require the effective shape.

The SET RANGE statement applied to a dummy argument has no effect on the effective range of the associated actual argument.

An example of a SET RANGE statement is:

```
REAL, RANGE / RLIST /, ARRAY (10, 10) :: A, B, C, D, E
N = 3
SET RANGE (N:2*N, N:2*N) / RLIST /
```

6.2.6 IDENTIFY Statement. The IDENTIFY statement provides a dynamic aliasing facility involving an alias object and a parent object. The scalar IDENTIFY statement provides a name-aliasing mechanism on a dynamic basis. The array IDENTIFY statement provides both a name-aliasing and remapping capability on a dynamic basis.

R623 *identify-stmt* **is** *identify-scalar-stmt*
or *identify-array-stmt*

6.2.6.1 Scalar IDENTIFY Statement.

R624 *identify-scalar-stmt* **is** IDENTIFY (*scalar-alias-name* = *parent*)

R625 *parent* **is** *scalar-variable-name*
or *array-element*
or *scalar-structure-component*
or *substring*

Constraint: The *scalar-alias-name* and the parent must conform in type and type parameters.

Constraint: The *scalar-alias-name* must have the ALIAS attribute and must not have the SAVE attribute.

Constraint: The *parent* must be a *variable*.

Constraint: The *scalar-alias-name* must not be the same as the name of the scalar parent, either directly or indirectly through multiple alias associations.

An alias variable is alias associated with a parent variable following a valid execution of an IDENTIFY statement. If the parent is an alias, it must be alias associated. An object that is alias associated must be directly or indirectly associated with a nonalias object. Rules governing the scope and association of aliases are described in Section 14.

If the same *scalar-alias-name* appears in more than one IDENTIFY statement, it must always have the same unqualified parent unless the parents are subobjects of the same named object.

An alias must not be referenced or defined unless it is alias associated with a parent that may be referenced or defined. If an alias is alias associated, it may be used according to the rules that govern the use of data objects except for certain restrictions (6.2.6.3).

The following are examples of scalar IDENTIFY statements:

(1) Scalar alias

```
IDENTIFY (NEW_NAME = SCALAR_VARIABLE)
IDENTIFY (ELT = B (K) )
IDENTIFY (CHUNK = STRING (5:22))
IDENTIFY (PART = STRUCTURE % COMPONENT )
```

(2) Dynamic scalar alias

```
IDENTIFY (PIECE = STRUCTURE % A (I) )
IDENTIFY (PIECE = STRUCTURE % A (J) )
IDENTIFY (PIECE = STRUCTURE % B )
```

All three statements in example (2) could be written in a single program unit.

6.2.6.2 Array IDENTIFY Statement. The array IDENTIFY statement establishes an element-by-element association between an alias-array object and an array-parent object. Such an alias has properties similar to those of an array section, but can specify a greater variety of subsets of the array elements of the parent. For example, an alias may be the diagonal of an array of rank two, or may have one subscript selecting elements of an array of derived type and another indexing a component of the array elements (Examples 1 and 2 below).

R626 *identify-array-stmt* **is** IDENTIFY (*alias-element* = ■
▒ *parent-array-element* [(*substring-range*)] , ■
▒ *alias-bounds-spec-list*)

R627 *alias-element* **is** *alias-name* (*subscript-name-list*)

R628 *parent-array-element* **is** *parent-array* (*mapping-subscript-list*)

R629 *alias-bound-spec* **is** *subscript-name* = *lower-bound* : *upper-bound*

Constraint: If *substring-range* is present, *parent-array-element* must be of type character. A *subscript-name* must not appear in a *substring-range* in the IDENTIFY statement.

Constraint: The *alias-element* and *parent-array-element* must conform in type and type parameters.

Constraint: Each subscript in a *parent-array-element* must be of a form such that each of the *alias-element subscript-name*s appears at most once, and each subscript must be linear in each of the *alias-element subscript-name*s. Thus, I+I would not be a valid subscript in a *parent-array-element* where I was an *alias-element subscript-name*. Each of the *alias-element subscript-name*s must appear at least once in a *subscript* in the *parent-array-element*.

Constraint: The *alias-element* must have the ALIAS attribute and must not have the SAVE attribute.

Constraint: The *parent-array-element* must be a *variable*.

Constraint: The number of *subscript-names* in an *alias-element* and the number of *alias-bound-specs* must equal the rank of the *alias-name*.

Constraint: Each *subscript-name* in the *subscript-name-list* must be identical to the *subscript-name* in the corresponding *alias-bounds-spec*. The *subscript-names* in both lists must appear in the same order. A *subscript-name* must not appear more than once in each list.

Constraint: A bound in an *alias-bound-spec* must not depend on any other data object or expression in the same IDENTIFY statement nor on any element of the alias object.

Constraint: The alias array elements established by alias association must all be associated with elements of the parent array such that each subscript of a parent array element is within the declared bounds for the corresponding dimension unless the size of that dimension of the alias array is zero.

Constraint: The *alias-element* name must not be the same as the name of the *parent-array-element*, either directly or indirectly through multiple alias associations.

An alias array is alias associated with a parent array following a valid execution of an IDENTIFY statement. An alias array must not be referenced or defined unless it is alias associated with a parent that may be referenced or defined. Note that an array that is alias associated must be directly or indirectly associated with a nonalias object. Rules governing the scope and association of aliases are described in Section 14.

Execution of an IDENTIFY statement for an alias array completely determines the declared bounds of the alias array. These are the bounds specified by *alias-bound-specs*. If the alias array also has the RANGE attribute, the effective bounds are set equal to the declared bounds.

The scope of the subscript names is the IDENTIFY statement itself, and the subscripts are implicitly of type integer. If a local or global data object has the same name, a reference to a data object with this name within the IDENTIFY statement is always a reference to the subscript name.

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

Each subscript expression in a *parent-array-element* subscript must be of the form described for a *mapping-subscript*.

R630 *mapping-subscript* **is** *scalar-int-expr*
or *subscript-map*

R631 *subscript-map* **is** [[*subscript-map*] *add-op*] *subscript-term*

R632 *subscript-term* **is** *add-operand* [* *subscript-name*]
or *subscript-name* [* *subscript-factor*]

R633 *subscript-factor* **is** [*subscript-factor* *] *mult-operand*

Constraint: If a *mapping-subscript* is a *scalar-int-expr*, it must not involve any *subscript-name*. If a *mapping-subscript* is a *subscript-map*, it must involve at least one *subscript-name*.

Constraint: An *add-operand* in a *subscript-term* must not involve any *subscript-name*.

Constraint: A *mult-operand* in a *subscript-factor* must not involve any *subscript-name*.

The mapping established by the IDENTIFY statement is unaffected by subsequent undefinition or redefinition of variables involved in subscript expressions in a *parent-array-element*. Two or more elements of an alias array must not be alias associated with the same datum.

If the same *alias-name* appears in more than one IDENTIFY statement, it must always have the same parent array unless the parent arrays are subobjects of the same named object. If the parent array is an alias array, it must be alias associated. If the parent array is an allocatable array, it must be currently allocated. Whenever an allocatable array is deallocated, all aliases associated with it cease to be alias associated.

If an alias array is alias associated, it may be used according to the rules that govern the use of data objects except for certain restrictions (6.2.6.3).

The following are examples of array IDENTIFY statements:

(1) Skew section

```
IDENTIFY (DIAG (I) = ARRAY (I, I), I = 1:N)
IDENTIFY (DIAG1 (I) = ARRAY2 (3, I, I), I = 1:N)
```

(2) Array of structure components

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

6.2.6.3 Alias Restrictions. Note that there are some restrictions on the use of alias objects. A specified precision or exponent range real or complex data object must not be associated with a default real or complex object in an IDENTIFY statement. An alias name must not appear in a namelist group object list, an EQUIVALENCE statement, as an object in a COMMON or SAVE statement, or as an array in an ALLOCATE or DEALLOCATE statement. The variables or arrays whose names are included in the *data-stmt-object-list* must not be alias objects. A SET RANGE statement must not be used to establish the effective range for an alias array that is not alias associated.

6.2.7 Summary of Array Name Appearances.

Table 6.2. Allowed Appearances of Array Names.

Place of Appearance	Explicit Shape Array	Alias Array	Structure Component Array	Allocatable Array
dummy-arg	Yes	No	No	Yes
use-stmt	Yes	Yes	No	Yes
type-declaration-stmt	Yes	Yes	No	Yes
namelist-stmt	Yes	No	No	No
equivalence-stmt	Yes	No	No	No
data-stmt	Yes	No	No	No
common-stmt	Yes	No	No	No
input-item-list or *output-item-list*	Yes	Yes	Yes	Yes
internal-file-unit	Yes	Yes	Yes	Yes
format	Yes	Yes	Yes	Yes
save-stmt	Yes	No	No	Yes
primary	Yes	Yes	Yes	Yes
assignment-stmt	Yes	Yes	Yes	Yes
identify-stmt as parent	Yes	Yes	Yes	Yes
identify-stmt as alias	No	Yes	No	No
allocate-stmt	No	No	No	Yes
deallocate-stmt	No	No	No	Yes

1 2	*actual-arg* in a reference to a procedure	Yes	Yes	Yes	Yes

7. EXPRESSIONS AND ASSIGNMENT

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

7.1. Expressions. An **expression** is 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. 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.

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).

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, 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. For convenience, the low-level operator construction rules, but not the constraints, have been duplicated below from Section 3 where appropriate. See 3.2.4 for the constraints on *defined-unary-op* (7.1.1.2) and *defined-binary-op* (7.1.1.7). The semantics are specified in 7.2 and 7.3.

7.1.1.1. Primary.

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

Examples of a *primary* are:

Example	Syntactic Class
1.0	*constant*
A	*variable*

`[1.0,2.0]`	*array-constructor*
`PERSON('Jones', 12)`	*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 7.9). Level-1 expressions are primaries optionally operated on by defined unary operators:

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

R321 *defined-unary-op* **is** . *letter* [*letter*]... .

Simple examples of a level-1 expression are:

Example	Syntactic Class
`A`	*primary* (R701)
`.INVERSE. B`	*level-1-expr* (R702)

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

```
.INVERSE. (A + 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*.

R703 *mult-operand* **is** *level-1-expr* [*power-op mult-operand*]

R704 *add-operand* **is** [*add-operand mult-op*] *mult-operand*

R705 *level-2-expr* **is** [*add-op*] *add-operand*
or *level-2-expr add-op add-operand*

R311 *power-op* **is** **

R312 *mult-op* **is** *
or /

R313 *add-op* **is** +
or –

Simple examples of a level-2 expression are:

Example	Syntactic Class	Remarks
`A`	*level-1-expr*	A is a *primary*. (R702)
`B ** C`	*mult-operand*	B is a *level-1-expr*, ** is a *power-op*, and C is a *mult-operand*. (R703)
`D * E`	*add-operand*	D is an *add-operand*, * is a *mult-op*, and E is a *mult-operand*. (R704)
`+1`	*level-2-expr*	+ is an *add-op* and 1 is an *add-operand*. (R705)
`F - I`	*level-2-expr*	F is a *level-2-expr*, – is an *add-op*, and I is an *add-operand*. (R705)

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*.

R706 *level-3-expr* **is** [*level-3-expr concat-op*] *level-2-expr*

R314 *concat-op* **is** //

Simple examples of a level-3 expression are:

Example	Syntactic Class
A	*level-2-expr* (R705)
B // C	*level-3-expr* (R706)

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*.

R707 *level-4-expr* **is** [*level-3-expr rel-op*] *level-3-expr*

R315 *rel-op* **is** .EQ.
or .NE.
or .LT.
or .LE.
or .GT.
or .GE.
or = =
or < >
or <
or < =
or >
or > =

Simple examples of a level-4 expression are:

Example	Syntactic Class
A	*level-3-expr* (R706)
B .EQ. C	*level-4-expr* (R707)
D < E	*level-4-expr* (R707)

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*.

R708 *and-operand* **is** [*not-op*] *level-4-expr*

R709 *or-operand* **is** [*or-operand and-op*] *and-operand*

R710 *equiv-operand* **is** [*equiv-operand or-op*] *or-operand*

R711 *level-5-expr* **is** [*level-5-expr equiv-op*] *equiv-operand*

R316 *not-op* **is** .NOT.

R317 *and-op* **is** .AND.

R318 *or-op* **is** .OR.

R319 *equiv-op* **is** .EQV.
or .NEQV.

Simple examples of a level-5 expression are:

Example	Syntactic Class
`A`	*level-4-expr* (R707)
`.NOT. B`	*and-operand* (R708)
`C .AND. D`	*or-operand* (R709)
`E .OR. F`	*equiv-operand* (R710)
`G .EQV. H`	*level-5-expr* (R711)
`S .NEQV. T`	*level-5-expr* (R711)

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.

R712 *expr* **is** [*expr defined-binary-op*] *level-5-expr*

R322 *defined-binary-op* **is** . *letter* [*letter*]... .

Simple examples of an expression are:

Example	Syntactic Class
`A`	*level-5-expr* (R711)
`B .UNION. C`	*expr* (R712)

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).

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

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

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

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.

Table 7.1. Type of Operands and Result for the Intrinsic Operation $[x_1]$ *op* x_2. (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.)

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

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 of the form *defined-unary-op* x_2 where there exists a function whose interface is explicit (12.3.1) in the scoping unit containing *defined-unary-op* x_2 that specifies the operation (7.3) for the operator *defined-unary-op*, or of the form *intrinsic-operator* x_2 where the type of x_2 is not that required for a unary intrinsic operation (7.1.2), and there exists a function whose interface is explicit in the scoping unit containing *intrinsic-operator* x_2 that specifies the operation for the operator *intrinsic-operator*.

A **defined binary operation** is an operation of the form x_1 *defined-binary-op* x_2 where there exists a function whose interface is explicit in the scoping unit containing x_1 *defined-binary-op* x_2 that specifies the operation (7.3) for the operator *defined-binary-op*, or of the form x_1 *intrinsic-operator* x_2 where the types or ranks of x_1 and x_2 are not those required for an intrinsic binary operation (7.1.2), and there exists a function whose interface is explicit in the scoping unit containing x_1 *intrinsic-operator* x_2 that specifies the operation for the operator *intrinsic-operator*.

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

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

An **extension operation** is a defined operation in which the operator is of the form *defined-unary-op* or *defined-binary-op*. Such an operator is called an **extension operator**. Note that the operator used in an extension operation may be overloaded in that more than one function whose interface is explicit in the scoping unit specifying the same operator may exist.

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

R713 *logical-expr* **is** *expr*

Constraint: *logical-expr* must be type logical.

R714 *char-expr* **is** *expr*

Constraint: *char-expr* must be type character.

R715 *int-expr* **is** *expr*

Constraint: *int-expr* must be type integer.

R716 *numeric-expr* **is** *expr*

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

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

7.1.4.1. Data Type, Type Parameters, and Shape of a Primary. The data type, type parameters, and shape of a primary are determined according to whether the primary is a constant, variable, array constructor, structure constructor, function reference, or parenthesized expression. If a primary is a constant, its type, type parameters, and shape are determined by the constant (4.3). If it is a structure constructor, it is scalar, its type is determined by the constructor name, and its type parameters are determined by the constructor type parameters (4.4.3). 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 determined from corresponding attributes of the variable (5.1.1, 5.1.2) or the function reference (12.4.2), respectively. Note that in the case of a function reference, the function may be generic (13.9) or overloaded (12.5.5), in which case its type, type parameters, and rank 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.

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

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

An expression whose type is real, complex, or character has type parameters. For an expression x_1 // x_2 where // is the intrinsic operator for character concatenation, the type parameter is the sum of the lengths of the operands. For an expression *op* x_2 where *op* is a numeric intrinsic unary operator and x_2 is of type real or complex, the type parameters of the expression are those of the operand. For an expression x_1 *op* x_2 where *op* is a numeric intrinsic binary operator with one operand of type integer and the other of type real or complex, the type parameters of the expression are those of the real or complex operand. In the case where both operands are any of type real or complex with type parameters p_1, r_1 and p_2, r_2,

where the p's are precision parameter values and the r's are exponent range parameter values, the type parameters of the expression are:

(1) Specified precision max (p_1,p_2) and specified exponent range max (r_1,r_2) if the operands have specified precisions and specified exponent ranges,

(2) Specified precision max (p_1,p_2) and unspecified exponent range if the operands have specified precisions and unspecified exponent ranges,

(3) Unspecified precision and specified exponent range max (r_1,r_2) if the operands have unspecified precisions and specified exponent ranges,

(4) Default real if both operands have default real parameters,

(5) Double precision if both operands have double precision parameters or one has default real parameters and the other has double precision parameters,

(6) Unspecifiable precision max (p_1,p_2) and exponent range max (r_1,r_2) in all other cases.

7.1.4.2.1. Unspecifiable Precision and Exponent Range Expression as Actual Argument. An expression with unspecifiable precision and exponent range may be an actual argument to a procedure only in the case where:

(1) The expression is the actual argument of an intrinsic function, or

(2) The expression is associated with a dummy argument whose type parameters are both asterisks.

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

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

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

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 operator is an intrinsic operator, and each primary is one of the following:

(1) A constant,

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

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

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

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

(6) A constant expression enclosed in parentheses.

R717 *constant-expr* **is** *expr*

R718 *char-constant-expr* **is** *char-expr*

R719 *int-constant-expr* **is** *int-expr*

R720 *logical-constant-expr* **is** *logical-expr*

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

The following are examples of constant expressions:

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

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

7.1.6.2 Type-Parameter Expression. A **nonprecision type-parameter expression** is a scalar integer expression in which no primary is a reference to a variable, a type parameter named PRECISION or EXPONENT_RANGE, a nonintrinsic function, or the intrinsic functions EFFECTIVE_PRECISION and EFFECTIVE_EXPONENT_RANGE.

7.1.6.2.1 Precision Expression. A **precision type-parameter expression** is a scalar integer expression whose primaries are:

(1) Integer constants, a reference to the intrinsic inquiry function EFFECTIVE_PRECISION with an argument that is a previously-defined component of the derived type, and similarly limited expressions enclosed in parentheses; or

(2) The dummy type-parameter name PRECISION.

7.1.6.2.2 Exponent Range Expression. An **exponent-range type-parameter expression** is a scalar integer expression whose primaries are:

(1) Integer constants, a reference to the intrinsic inquiry function EFFECTIVE_EXPONENT_RANGE with an argument that is a previously defined component of the derived type, and similarly limited expressions enclosed in parentheses; or

(2) The dummy type-parameter name EXPONENT_RANGE.

Examples of nonprecision type parameter expressions are:

```
10
P + Q
LEN (X)
```

where P and Q are type parameters and X is a previously declared character component.

Examples of precision type parameter expressions are:

```
10
PRECISION
EFFECTIVE_PRECISION (Y)
2 * PRECISION
```

where Y is a previously declared component having a precision parameter.

Examples of exponent range type parameter expressions are:

```
10
EXPONENT_RANGE
EFFECTIVE_EXPONENT_RANGE (Y)
2 * EXPONENT_RANGE
```

where Y is a previously declared component having an exponent range parameter. It is a requirement that if the precision type parameter expression is PRECISION for a component, the exponent range type parameter expression for the component must be EXPONENT_RANGE and vice versa (4.4.1.1).

7.1.6.3 Specification Expression. A **restricted expression** is an expression in which each primary is:

(1) A constant,

(2) A variable that is a dummy argument,

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

(4) A variable that is made accessible by use or host association,

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

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

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

(8) A restricted expression enclosed in parentheses.

R721 *specification-expr* **is** *scalar-int-expr*

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

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 sequence, the type parameter or array bound must be specified in a prior specification expression of the specification sequence.

The following are examples of specification expressions:

```
DLBOUND (B, 1) + 5
M + LEN (C)
2 * EFFECTIVE_PRECISION (A)
```

where B, M, and C are dummy arguments, B is an assumed-shape array, and 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 must be defined at the time the reference is executed. An integer operand must be defined with an integer value rather than a statement label value. All of the characters in a character data object reference must be defined.

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

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

The execution of a function reference must not alter the value of any other variable within the statement in which the function reference appears. The execution of a function reference in a statement must not define or redefine (14.8) the value of any variable in common (5.5.2) or any variable made accessible by use or host association (11.2.2, 11.3.2) if the definition or redefinition affects the value of any other reference in the statement. However, execution of a function reference in the logical expression of an IF statement (8.1.2.4) or WHERE statement (7.5.2.1) is permitted to define variables in the statement that is executed when the value of the expression is true. For example, in the statements:

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

F or G may define X. If a function reference causes definition or undefinition of an actual argument of the function, that argument or any associated entities must not appear elsewhere in the same statement. For example, the statements

```
A (I) = F (I)
Y = G (X) + X
```

are prohibited if the reference to F defines or undefines I or the reference to G defines or undefines 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, except that the result of a function may assume a type that depends on the type of its arguments as specified in Sections 12 and 13.

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

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

Execution of an array section reference requires the evaluation of its section subscripts. 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. Note that 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.

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

```
A + B
```

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

When an intrinsic unary operator operates on 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 if the value of the expression can be determined otherwise. This principle is most often applicable to logical expressions and zero-sized arrays, but it applies to all expressions. For example, in evaluating the expression

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

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

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

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

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

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

Two expressions of a numeric type are mathematically equivalent if, for all possible values of their primaries, their mathematical values are equal. However, mathematically equivalent expressions of numeric type may produce different computational results. For example, any difference between the values of the expressions (1./3.)*3. and 1. is a computational difference, not a mathematical difference.

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

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

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

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

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

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

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

the parenthesized expression (B – C) must be evaluated and then added to A.

Note that the inclusion of parentheses may change the mathematical value of an expression. For example, the two expressions:

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

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

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

```
Z + R + I
```

where Z, R, and I represent 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 a character intrinsic operation. A processor needs to evaluate only as much of the character intrinsic operation as is required by the context in which the expression appears. For example, the statements

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

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

7.1.7.5. 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. 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 is not violated. For example, for the variables L1, L2, and L3 of type logical, the processor may choose to evaluate the expression

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

as

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

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

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

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

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

Table 7.2. Interpretation of the Numeric Intrinsic Operators.

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

The interpretation of a division 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). For example, 2**(–3) has the value of 1/(2**3), which is zero.

7.2.1.1. Integer Division. One operand of type integer may be divided by another operand of type integer. Although the mathematical quotient of two integers is not necessarily an integer, Table 7.1 specifies that an expression involving the division operator with two operands of type integer is interpreted as an expression of type integer. The result of such an operation is the integer closest to the mathematical quotient and between zero and the

mathematical quotient inclusively. For example, the expression (−8)/3 has the value (−2).

7.2.1.2. Complex Exponentiation. In the case of a complex value raised to a complex power, the value of the operation is the "principal value" determined by x_1 ** x_2 = EXP(x_2 * LOG(x_1)), where EXP and LOG are functions described in 13.12.

7.2.2. Character Intrinsic Operation. The character intrinsic operator // is used to concatenate two operands of type character. Evaluation of the character intrinsic operation produces a result of type character.

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

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

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

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

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

Evaluation of a relational intrinsic operation produces a result of type logical.

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

Table 7.6. Interpretation of the Relational Intrinsic Operators.

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

< >	Not Equal To	$x_1 <> x_2$	x_1 not equal to x_2

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

The value of the relational operation

x_1 *rel-op* x_2

is the value of the expression

$((x_1) - (x_2))$ rel−op 0

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

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

For a character relational intrinsic operation, the operands are compared one character at a time in order, beginning with the first character of each character operand. 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 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 (3.1.6); 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 that the collating sequence depends partially on the processor; however, the result of the use of the operators .EQ., .NE., = =, and < > does not depend on the collating sequence.

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

Table 7.7. Interpretation of the Logical Intrinsic Operators.

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

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

Table 7.8. The Values of Operations Involving Logical Intrinsic Operators

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

1	true	true	false	true	true	true	false
2	true	false	true	false	true	false	true
3	false	true	false	false	true	false	true
4	false	false	true	false	false	true	false

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

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

(1) The function subprogram is specified with a FUNCTION statement (12.5.2.2) that specifies one dummy argument d_2 and has a suffix that includes OPERATOR *(op)*,

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

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

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

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

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

(1) The function subprogram is specified with a FUNCTION statement (12.5.2.2) that specifies two dummy arguments, d_1 and d_2, and has a suffix that includes OPERATOR *(op)*,

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

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

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

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

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

Table 7.9. Categories of Operations and Relative Precedences.

Category of Operation	Operators	Precedence
Extension	*defined-unary-op*	Highest
Numeric	**	.
Numeric	* or /	.
Numeric	unary + or −	.
Numeric	binary + or −	.
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, whether it is an overloaded intrinsic operator or an extension operator.

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. For example, 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 (7.1.1). 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'
```

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

```
A * .INVERSE. B
```

As another example, in the expression

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

the general form (7.1.1) implies a higher precedence for the .AND. operator than the .OR. operator; therefore, the interpretation of the above expression is the same as the interpretation of the expression

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

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

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

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

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

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

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

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

7.5.1.1. General Form.

R722 *assignment-stmt* **is** *variable* = *expr*

where *variable* is defined in 2.4.5 and *expr* is defined in 7.1.1.7.

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 effective shapes of *variable* and *expr* conform and where:

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

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

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

An **array intrinsic assignment statement** is an intrinsic assignment statement for which *variable* is an array.

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

Type of *variable*	Type of *expr*
integer	integer, real, complex
real	integer, real, complex
complex	integer, real, complex
character	character
logical	logical
derived type	same derived type and type parameter values 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 whose interface is explicit (7.5.1.6).

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

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

Table 7.11. Numeric Conversion and Assignment Statement *variable* = *expr*

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

(The functions INT, REAL, DBLE, and CMPLX are the generic functions defined in 13.12.)

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

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

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

7.5.1.5. Interpretation of Intrinsic Assignments. Execution of an intrinsic assignment causes, in effect, the evaluation of the expression *expr* and all expressions within *variable* (7.1.7), the possible conversion of *expr* to the type and type parameters of *variable* (Table 7.11), and the definition of *variable* with the resulting value. The execution of the assignment must appear as if the evaluation of all operations in *expr* and, if present, all operations in the subscripts or section subscripts of *variable* occurred before any portion of *variable* is defined by the assignment. The evaluation of expressions within *variable* must neither affect nor be affected by the evaluation of *expr*.

Both *variable* and *expr* may contain references to any portion of *variable*. 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). That is, if the value of STRING prior to the assignment was 'ABCDEF', the value following the assignment is 'AABCDF'.

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

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

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)
```

An intrinsic derived-type assignment is performed as if each component of *expr* were assigned to the corresponding component of *variable* using intrinsic assignment or derived-type assignment statements, where accessible. The type parameters, if any, of *variable* and *expr* may be different. The processor may perform the component-by-component assignment in any order or by any means that has the same effect.

For an example of a derived-type intrinsic assignment statement, if C and D are of the same derived type with components S, T, U, and V of type integer, logical, character, and another derived type, respectively, the intrinsic assignment.

```
C = D
```

assigns D % S to C % S using the numeric 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 subprogram that defines the operation.

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

(1) The subroutine subprogram is specified with a SUBROUTINE statement (12.5.2.3) that specifies two dummy arguments, d_1 and d_2, and has ASSIGNMENT specified,

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

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

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

(5) d_1 and d_2 are scalar and x_1 and x_2 are in shape conformance for intrinsic assignment (7.5.1.4), or d_1 or d_2 (or both) is an array and the effective shapes of x_1 and x_2 match those of d_1 and d_2, respectively.

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

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

R723 *where-stmt* **is** WHERE (*mask-expr*) *assignment-stmt*

R724 *where-construct* **is** *where-construct-stmt*
[*assignment-stmt*]...
[*elsewhere-stmt*
[*assignment-stmt*]...]
end-where-stmt

R725 *where-construct-stmt* **is** WHERE (*mask-expr*)

R726 *mask-expr* **is** *logical-expr*

R727 *elsewhere-stmt* **is** ELSEWHERE

R728 *end-where-stmt* **is** END WHERE

Constraint: In each *assignment-stmt*, the *mask-expr* and the variable being defined must be arrays of the same effective shape.

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
```

7.5.2.2. Interpretation of Masked Array Assignments. When the *assignment-stmt* in a *where-stmt* is executed, the *expr* is evaluated for all the elements where *mask-expr* is true and the result is assigned to the corresponding elements of *variable*. When a *where-block* is executed, the *mask-expr* is evaluated and the result saved by the processor. Each *assignment-stmt* in the where block is evaluated, in sequence, as if it were WHERE (*expr*) *assignment-stmt* and each *assignment-stmt* in the ELSEWHERE block is evaluated, in sequence, as if it were WHERE (.NOT. *expr*) *assignment-stmt*.

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

If an elemental function reference occurs in *expr* and is not an actual argument of a nonelemental function reference, the function is evaluated only for the elements corresponding to true values in *mask-expr* (false values after ELSEWHERE).

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

8 EXECUTION CONTROL

Control constructs are used to control the execution sequence. These constructs include executable constructs containing blocks and executable statements that may be used to alter the execution sequence.

8.1 Executable Constructs Containing Blocks. The following are executable constructs that contain blocks and may be used to control the execution sequence:

(1) IF Construct

(2) CASE Construct

(3) DO Construct

The *do-body* of the DO construct is a block only if the *do-termination* is an END DO statement.

A **block** is a sequence of executable constructs that is treated as an integral 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. Note that a block may be empty.

Any of these three constructs may be named. If a construct is named, the name must be the first lexical token of the first statement of the construct and the last lexical token of the construct. In fixed form, the name preceding the construct must be placed after column 6.

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 must be entirely contained within the block.

8.1.1.2 Control Flow in Blocks. Transfer of control to the interior of a block from outside the block is prohibited. Transfers within a block and transfers from the interior of a block to outside the block may occur. For example, if a statement inside the block has a statement label, a GO TO statement using that label may appear in the same block. Subroutine and function references may appear in a block (12.4.2, 12.4.4).

8.1.1.3 Execution of a Block. Execution of a block begins with the execution of the first executable construct in the block. Unless there is a transfer of control out of the block, the execution of the block is completed when the last executable construct in the sequence is executed. The action that takes place at the terminal boundary depends on the 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.

8.1.2.1 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 an *if-construct-name* is present, the same name must be specified on both the *if-then-stmt* and the corresponding *end-if-stmt*. If an *if-construct-name* appears on the *if-then-stmt*, it may also optionally appear on any *else-if-stmt* or *else-stmt* belonging to that *if-construct*.

8.1.2.2 Execution of an IF Construct. At most one of the blocks contained within the IF construct is executed. If there is an ELSE statement in the construct, exactly one of the blocks contained within the construct will be executed. The 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 expressions in any remaining ELSE IF statements of the IF construct are not evaluated. If none of the evaluated expressions are true and there is no ELSE statement, the execution of the construct is completed without the execution of any blocks within the construct.

An ELSE IF statement or an ELSE statement must not be a branch target statement. It is permissible to branch to an END IF statement from within the IF construct, and also from outside the construct.

8.1.2.3 Examples of IF Constructs.

```
IF (CVAR .EQ. 'RESET') THEN
   I = 0; J = 0; K = 0
END IF
IF (PROP) THEN
   WRITE (3, '("QED")')
   STOP
ELSE
   PROP = NEXTPROP
END IF

IF (A .GT. 0) THEN
   B = C/A
   IF (B .GT. 0) THEN
      D = 1.0
   END IF
ELSE IF (C .GT. 0) THEN
   B = A/C
   D = -1.0
ELSE
   B = ABS (MAX (A, C))
   D = 0
END IF
```

8.1.2.4 IF Statement. The IF statement controls a single action statement (R223).

R807 *if-stmt* **is** IF (*scalar-logical-expr*) *action-stmt*

Constraint: The *action-stmt* in the *if-stmt* must not be an *if-stmt*.

Execution of an IF statement causes evaluation of the scalar 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 is permitted to affect entities in the action statement.

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 exactly one of its constituent blocks.

8.1.3.1. Form of the CASE Construct.

R808 *case-construct* **is** *select-case-stmt*
[*case-stmt*
block]...
end-select-stmt

R809 *select-case-stmt* **is** [*select-construct-name* :] SELECT CASE (*case-expr*)

R810 *case-stmt* **is** CASE *case-selector* [*select-construct-name*]

R811 *end-select-stmt* **is** END SELECT [*select-construct-name*]

Constraint: If a *select-construct-name* is present, the same name must be specified on both the *select-case-stmt* and the corresponding *end-select-stmt*. If a *select-construct-name* appears on the *select-case-stmt*, it may also optionally appear on any *case-stmt* belonging to that *case-construct*.

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: Only one DEFAULT *case-selector* may appear in any given *case-construct*.

R814 *case-value-range* **is** *case-value*
or *case-value* :
or : *case-value*
or *case-value* : *case-value*

R815 *case-value* **is** *scalar-int-constant-expr*
or *scalar-char-constant-expr*
or *scalar-logical-constant-expr*

Constraint: For a given CASE construct, each *case-value* must be of the same type as *case-expr*. For character type, length differences are allowed.

Constraint: A *case-value-range* using a colon must not be used if *case-expr* is of type logical.

8.1.3.2. Execution of a CASE Construct. The execution of the SELECT CASE statement causes the case expression to be evaluated. The resulting value is called the **case index** and must match exactly one of the selectors of one of the CASE statements of the construct. For a case value range list, a match occurs if the case index matches any of the case value ranges in the list. For a case index with a value of *c*, a match is determined as follows:

(1) If the case value range contains a single value *v* without a colon, a match occurs for data type logical if the expression *c* .EQV. *v* is true. A match occurs for data type integer 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, a DEFAULT selector must be present, and it matches the case index.

The block following the CASE statement containing the matching selector is executed. This completes execution of the construct.

One and only one of the blocks of a CASE construct is executed.

The case value ranges in different selectors must not overlap; that is, there must be no possible value of the case index that matches more than one selector. Case value ranges within a single case selector may overlap.

A CASE statement must 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. An integer signum function:

```
INTEGER FUNCTION SIGNUM (N)
SELECT CASE (N)
CASE (:-1)
   SIGNUM = -1
CASE (0)
   SIGNUM = 0
CASE (1:)
   SIGNUM = 1
END SELECT
END
```

A code fragment to check for balanced parentheses:

```
CHARACTER LINE (80)
...
LEVEL=0
DO I = 1, 80
   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
```

The following three fragments are equivalent:

```
IF (SILLY .EQ. 1) THEN
   CALL THIS
ELSE
   CALL THAT
END IF

SELECT CASE (SILLY .EQ. 1)
CASE (.TRUE.)
   CALL THIS
CASE (.FALSE.)
   CALL THAT
END SELECT

SELECT CASE (SILLY)
CASE DEFAULT
   CALL THAT
CASE (1)
   CALL THIS
END SELECT
```

8.1.4. Iteration Control. The DO construct is used to provide iteration control by specifying the repeated execution of a sequence of executable constructs.

8.1.4.1. Form of the DO Construct.

R816 *do-construct* **is** *do-stmt*
do-body
do-termination

R817 *do-stmt* **is** [*do-construct-name* :] DO [*label*] [*loop-control*]

R818 *loop-control* **is** [,] *do-variable* = *scalar-numeric-expr*, ■
■ *scalar-numeric-expr* [, *scalar-numeric-expr*]
or (*scalar-int-expr*) TIMES

R819 *do-variable* **is** *scalar-variable*

Constraint: The *do-variable* must be a scalar integer, default real, or default double precision real named variable.

Constraint: Each *scalar-numeric-expr* in *loop-control* must be of type integer, default real, or default double precision real.

R820 *do-body* **is** [*execution-part-construct*]...

R821 *do-termination* **is** *end-do-stmt*
or *continue-stmt*
or do-term-stmt
or do-construct

R822 *do-term-stmt* is action-stmt

Constraint: If the *label* is omitted in a *do-stmt*, the corresponding *do-termination* must be an *end-do-stmt*.

Constraint: If a *label* appears in the *do-stmt* and the corresponding *do-termination* is not a *do-construct*, the *do-termination* must be identified with that label.

Constraint: If the *do-termination* is a *continue-stmt* or *do-term-stmt*, the corresponding *do-stmt* must contain a label.

Constraint: A *do-term-stmt* must not be a *continue-stmt*, *goto-stmt*, *return-stmt*, *stop-stmt*, *exit-stmt*, *cycle-stmt*, *arithmetic-if-stmt*, nor *assigned-goto-stmt*.

Constraint: If the *do-termination* is a *do-construct*, both of the corresponding *do-stmt*s must specify the same label.

Constraint: If a *do-termination* is a *do-construct*, the *do-termination* of that *do-construct* must not be an *end-do-stmt*.

R823 *end-do-stmt* **is** END DO [*do-construct-name*]

Constraint: If a *do-construct-name* is used on the *do-stmt*, the corresponding *do-termination* must be an *end-do-stmt* that uses the same *do-construct-name*. If a *do-construct-name* does not appear on the *do-stmt*, a *do-construct-name* must not appear on the corresponding *end-do-stmt*.

R824 *exit-stmt* **is** EXIT [*do-construct-name*]

R825 *cycle-stmt* **is** CYCLE [*do-construct-name*]

Constraint: An *exit-stmt* or a *cycle-stmt* must be within the range of one or more *do-constructs*.

Constraint: An *exit-stmt* or *cycle-stmt* using a *do-construct-name* must be within the range of the *do-construct* that has that name.

An EXIT statement or CYCLE statement is said to **belong** to a specific DO construct. If the EXIT statement or CYCLE statement contains a construct name, it belongs to the DO construct using that name. Otherwise, it belongs to the innermost DO construct in which it appears.

8.1.4.2. Range of a DO Construct. The **range** of a DO construct consists of the *do-body* and the *continue-stmt*, *do-term-stmt*, or terminating *do-construct*, if any. The range must satisfy the rules for blocks (8.1.1). Note that if the *do-termination* is an END DO statement, the range is a block (8.1). If the *do-termination* is a *continue-stmt*, *do-term-stmt*, or *do-construct*, a terminal boundary delimiting the range is assumed (8.1.1.3).

Within a scoping unit, all DO constructs whose DO statements use the same label are said to **share** the statement identified with that label. Note that the statement so identified must be a CONTINUE statement or *do-term-stmt* that serves as the do termination of the innermost of these DO constructs.

It is permissible to branch to an END DO statement only from within the range of the DO construct that it terminates. A transfer of control to a statement within the range of a DO construct from outside the range is prohibited.

8.1.4.3. Active and Inactive DO Constructs. A DO construct is either **active** or **inactive**. Initially inactive, a DO construct becomes active only when its DO statement is executed.

Once active, the DO construct becomes inactive only when the construct it specifies is terminated (8.1.4.4.4).

When a DO construct becomes inactive, the *do-variable*, if any, retains its last defined value.

8.1.4.4. Execution of a DO Construct. A DO construct specifies a loop. A **loop** is a sequence of executable constructs that is executed repeatedly. There are three phases in the execution of a DO construct: initiation of the loop, execution of the loop body, and termination of the loop.

8.1.4.4.1. Loop Initiation. When the DO statement is executed, the DO construct becomes active. If there is *loop-control* of the form *do-variable* = *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 established by evaluating *scalar-numeric-expr*$_1$, *scalar-numeric-expr*$_2$, and *scalar-numeric-expr*$_3$, respectively, including, if necessary, conversion to the type of the *do-variable* according to the rules for numeric conversion (Table 7.11).

If *scalar-numeric-expr*$_3$ does not appear, m_3 has a value of one. m_3 must not have a value of zero.

(2) The DO variable becomes defined with the value of the initial parameter m_1.

(3) The **iteration count** is established and is the value of the expression

$$\text{MAX (INT } ((m_2 - m_1 + m_3) / m_3), 0)$$

Note that the iteration count is zero whenever:

$m_1 > m_2$ and $m_3 > 0$, or
$m_1 < m_2$ and $m_3 < 0$.

If *loop-control* takes the form (*scalar-int-expr*) TIMES, the *scalar-int-expr* is evaluated. If the resulting value is positive, it becomes the iteration count; otherwise, the iteration count is zero.

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.

At the completion of the execution of the DO statement, the execution cycle begins.

8.1.4.4.2. The Execution Cycle. The **execution cycle** of a DO construct consists of the following steps performed in sequence:

(1) The iteration count, if any, is tested. If the iteration count is zero, the *do-construct* becomes inactive. If, as a result, all of the *do-constructs* sharing the *do-term-stmt* or *continue-stmt* are inactive, the execution of the construct is complete. However, if some of the DO constructs sharing the *do-term-stmt* or *continue-stmt* are active, execution continues with step (3) of the execution cycle of the active DO construct whose DO statement was most recently executed.

(2) If the iteration count is nonzero, the range of the DO construct is executed.

(3) The iteration count, if any, is decremented by one. The *do-variable*, if any, is incremented by the value of the incrementation parameter m_3.

(4) This cycle is executed repeatedly from step (1) until the loop is terminated.

Except for the incrementation of the DO variable that occurs in step (3), the DO variable must neither be redefined nor become undefined while the DO construct is active. If the *do-termination* is included within the range of the DO construct (8.1.4.2), execution of the *do-termination* occurs as a result of the normal execution sequence or as a result of a transfer of control from within the range of the DO construct. Unless execution of the *do-term-stmt*, if any, results in a transfer of control, execution continues with step (3) of the execution cycle.

8.1.4.4.3. Cycle Interruption. Execution of a CYCLE statement that belongs to a DO construct causes immediate execution of step (3) of the current execution cycle of that DO construct. A transfer of control to an END DO statement has the same effect as a CYCLE statement that belongs to the DO construct it terminates.

8.1.4.4.4. Loop Termination. The execution of the loop is complete when one of the following conditions occurs:

(1) The iteration count, tested during step (1) of the execution cycle, is determined to be zero.

(2) An EXIT statement that belongs to the DO construct is executed.

(3) An EXIT statement or a CYCLE statement that is contained in the DO construct but belongs to another DO construct containing this one is executed.

(4) A RETURN statement within the range is executed.

(5) Control is transferred to a statement that is neither the *do-termination* nor within the range of the DO construct.

(6) A STOP statement anywhere in the program is executed, or execution is terminated for any other reason.

8.1.4.5. Examples of DO Constructs.

Example 1:

```
DO
   IF (X .GT. Y) THEN
      Z = X
      EXIT
   END IF
   CALL NEWX (X)
END DO
```

The above program fragment contains a DO construct that does not have an iteration count. The loop will continue to execute until X becomes greater than Y, at which point the EXIT statement terminates the loop.

Example 2:

```
SUM = 0
READ *, N
DO (N) TIMES
   READ *, P, Q
   CALL CALCULATE (P, Q, R)
   SUM = SUM + R
   IF (SUM .GT. SMAX) EXIT
END DO
```

The loop is executed N times unless SUM becomes greater than SMAX. If N is set to 0 by the READ statement, the loop is not executed.

Example 3:

```
N = 0
DO I = 1, 10
   J = I
   DO K = 1, 5
      L = K
      N = N + 1  ! THIS STATEMENT EXECUTES 50 TIMES.
   END DO
END DO
```

After execution of the above program fragment, I = 11, J = 10, K = 6, L = 5, and N = 50.

Example 4:

```
N = 0
DO I = 1, 10
   J = I
   DO K = 5, 1  ! THIS INNER LOOP IS NOT EXECUTED.
      L = K
      N = N + 1
   END DO
END DO
```

After execution of the above program fragment, I=11, J=10, K=5, N=0, and L is not defined.

Example 5:

```
      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.
```

After execution of the above statements, I = 11, J = 10, K = 6, L = 5, and N = 50.

Example 6:

```
       N = 0
       DO 200 I = 1, 10
          J = I
          DO 200 K = 5, 1  ! THIS INNER LOOP IS NOT EXECUTED
             L = K
200          N = N + 1
```

After execution of the above statements, I = 11, J = 10, K = 5, N = 0, and L is not defined.

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 *end-program-stmt*, an *end-function-stmt*, an *end-subroutine-stmt*, an *if-then-stmt*, an *end-if-stmt*, a *select-stmt*, an *end-select-stmt*, a *do-stmt*, a *do-termination*, 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 from within its IF construct, and also from outside the construct.

It is permissible to branch to a DO termination only from within its DO construct.

8.2.1. Statement Labels. **Statement labels** provide a means of referring to individual statements. Any statement not forming part of another statement may be identified with a label, but only branch target statements, FORMAT statements, and DO terminations may be referred to by the use of statement labels (3.2.5).

8.2.2. GO TO Statement.

R826 *goto-stmt* **is** GO TO *label*

Constraint: *label* must 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.

R827 *computed-goto-stmt* **is** GO TO (*label-list*) [,] *scalar-int-expr*

Constraint: Each *label* in *label-list* must be the statement label of a branch target 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 $1 \leq i \leq n$ where n is the number of labels in *label-list*, a transfer of control occurs so that the next statement executed is the one identified by the ith label in the list of labels. If i is less than 1 or greater than n, the execution sequence continues as though a CONTINUE statement were executed.

8.2.4. ASSIGN and Assigned GO TO Statement.

R828 *assign-stmt* **is** ASSIGN *label* TO *scalar-int-variable*

Constraint: *label* must be the statement label of a branch target statement or a *format-stmt*.

R829 *assigned-goto-stmt* **is** GO TO *scalar-int-variable* [[,] (*label-list*)]

Constraint: Each *label* in *label-list* must be the statement label of a branch target statement that appears in the same scoping unit as the *assigned-goto-stmt*.

Execution of an ASSIGN statement causes a statement label to be assigned to an integer variable. The statement label must be the label of a statement that appears in the same scoping unit as the ASSIGN statement. A label may appear more than once in the label list.

When an input/output statement containing the integer variable as a format specifier (9.4.1.1) is executed, the integer variable must be defined with the label of a FORMAT statement. While defined with a statement label value, the integer variable must not be referenced in any other context.

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

At the time of execution of an assigned GO TO statement, the integer variable must be defined with the value of a statement label of a branch target 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 must be one of the statement labels in the list.

8.2.5. Arithmetic IF Statement.

R830 *arithmetic-if-stmt* **is** IF (*scalar-numeric-expr*) *label*, *label*, *label*

Constraint: Each *label* must 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* must 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.

R831 *continue-stmt* **is** CONTINUE

CONTINUE statements are usually identified by labels that also appear in control statements, such as the DO statement.

8.4. STOP Statement.

R832 *stop-stmt* **is** STOP [*stop-code*]

R833 *stop-code* **is** *scalar-char-constant*
or *digit* [*digit* [*digit* [*digit* [*digit*]]]]

Execution of a STOP statement causes termination of execution of the executable program. At the time of termination, the stop code if any, is accessible. Leading zero digits are not significant.

8.5. PAUSE Statement.

R834 *pause-stmt* is PAUSE [stop-code]

Execution of a PAUSE statement causes a suspension of execution of the executable program. Execution must be resumable. At the time of suspension of execution, the stop code, if any, is accessible. Resumption of execution is not under control of the program. If execution is resumed, the execution sequence continues as though a CONTINUE statement were executed. Leading zero digits in the stop code are not significant.

9. INPUT/OUTPUT STATEMENTS

Input statements provide the means of transferring data from external media to internal storage or from an internal file to internal storage. This process is called **reading**. **Output statements** provide the means of transferring data from internal storage to external media or from internal storage to an internal file. This process is called **writing**. Some input/output statements specify that editing of the data is to be performed.

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

The input/output statements are the OPEN, CLOSE, READ, WRITE, PRINT, BACKSPACE, ENDFILE, REWIND, and INQUIRE statements.

The READ statement is a **data transfer input statement**. The WRITE statement and the PRINT statement are **data transfer output statements**. The OPEN statement and the CLOSE statement are **file connection statements**. The INQUIRE statement is a **file inquiry statement**. The BACKSPACE, ENDFILE, and REWIND statements are **file positioning statements**.

9.1. Records. A **record** is a sequence of values or a sequence of characters. For example, a line on a terminal is usually considered to be a record. However, a record does not necessarily correspond to a physical entity. There are three kinds of records:

(1) Formatted

(2) Unformatted

(3) Endfile

9.1.1. Formatted Record. A **formatted record** consists of a sequence of characters that are capable of representation in the processor. The length of a formatted record is measured in characters and depends primarily on the number of characters put into the record when it is written. However, it may depend on the processor and the external medium. The length may be zero. Formatted records may be read or written only by formatted input/output statements.

Formatted records may be prepared by means other than Fortran; for example, by some manual input device.

9.1.2. Unformatted Record. An **unformatted record** consists of a sequence of values in a processor-dependent form and may contain 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 input/output list (9.4.2) used when it is written, as well as on the processor and the external medium. The length may be zero. Unformatted records may be read or written only by unformatted input/output statements.

9.1.3. Endfile Record. An **endfile record** is written explicitly by the ENDFILE statement. The file must be connected for sequential access. An endfile record is written implicitly to a file connected for sequential access when the last operation on the file is an output statement other than the ENDFILE statement, and:

(1) A REWIND or BACKSPACE statement references the unit, or

(2) The unit (file) is closed, either explicitly by a CLOSE statement or implicitly by a program termination not caused by an error condition.

An endfile record may occur only as the last record of a file. An endfile record does not have a length property.

9.2. Files. A **file** is a sequence of records.

There are two kinds of files:

(1) External

(2) Internal

9.2.1. External Files. An **external file** is any file that exists in a medium external to the executable program.

The records of a file are either all formatted or all unformatted, except that the last record of a file may be an endfile record. At any given time, there is a processor-determined **set of allowed access methods**, a processor-determined **set of allowed forms**, and a processor-determined **set of allowed record lengths** for a file.

A file may have a name; a file that has a name is called a **named file**. The name of a named file is a character string. The set of allowable names 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-determined set of external files that are said to **exist** for an executable program. A file may be known to the processor, yet not exist for an executable program at a particular time. For example, there may be security reasons that prevent a file from existing for an executable program. A file may exist and contain no records; an example is a newly created file not yet written.

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

All input/output statements may refer to files that exist. An INQUIRE, OPEN, CLOSE, WRITE, PRINT, REWIND, or ENDFILE statement may also refer to a file that does not exist.

9.2.1.2. File Access. There are two methods of accessing the records of an external file, sequential and direct. Some files may have more than one allowed access method; other files may be restricted to one access method. For example, a processor may allow only sequential access to a file on magnetic tape. Thus, the set of allowed access methods depends on the file and the processor.

The method of accessing the file is determined when the file is connected to a unit (9.3.2).

9.2.1.2.1. Sequential Access. When connected for sequential access, an external file has the following properties:

(1) The order of the records is the order in which they were written if the direct access method is not a member of the set of allowed access methods for the file. If the direct access method is also a member of the set of allowed access methods for the file, the order of the records is the same as that specified for direct access. In this case, the first record accessed by sequential access is the record whose record number is 1 for direct access. The second record accessed by sequential access is the record whose record number is 2 for direct access, etc. A record that has not been written since the file was created must not be read.

(2) The records of the file are either all formatted or all unformatted, except that the last record of the file may be an endfile record. Unless the previous operation on the file was an output statement, the last record, if any, of the file must be an endfile record.

(3) The records of the file must not be read or written by direct access input/output statements.

9.2.1.2.2. Direct Access. When connected for direct access, an external file has the following properties:

(1) Each record of the file is uniquely identified by a positive integer called the **record number**. The record number of a record is specified when the record is written. Once established, the record number of a record can never be changed. Note that a record may not be deleted; however, a record may be rewritten. The order of the records is the order of their record numbers.

(2) The records of the file are either all formatted or all unformatted. If the sequential access method is also a member of the set of allowed access methods for the file, its endfile record, if any, is not considered to be part of the file while it is connected for direct access. If the sequential access method is not a member of the set of allowed access methods for the file, the file must not contain an endfile record.

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

(4) All records of the file have the same length.

(5) Records need not be read or written in the order of their record numbers. Any record may be written into the file while it is connected to a unit. For example, it is permissible to write record 3, even though records 1 and 2 have not been written. Any record may be read from the file while it is connected to a unit, provided that the record has been written since the file was created.

(6) The records of the file must not be read or written using list-directed (10.8) or namelist formatting (10.9).

9.2.1.3. File Position. Execution of certain input/output statements affects the position of a file. Certain circumstances can cause the position of a file to become indeterminate.

The **initial point** of a file is the position just before the first record. The **terminal point** is the position just after the last record.

If a file is positioned within a record, that record is the **current record**; otherwise, there is no current record.

Let n be the number of records in the file. If $1 < i \leq n$ and a file is positioned within the ith record or between the $(i-1)$th record and the ith record, the $(i-1)$th record is the **preceding record**. If $n \geq 1$ and the file is positioned at its terminal point, the preceding record is the nth and last record. If $n = 0$ or if a file is positioned at its initial point or within the first record, there is no preceding record.

If $1 \leq i < n$ and a file is positioned within the ith record or between the ith and $(i+1)$th record, the $(i+1)$th record is the **next record**. If $n \geq 1$ and the file is positioned at its initial point, the first record is the next record. If $n = 0$ or if a file is positioned at its terminal point or within the nth (last) record, there is no next record.

9.2.1.3.1. File Position Prior to Data Transfer. The positioning of the file prior to data transfer depends on the method of access: sequential or direct.

For sequential access on input, the file is positioned at the beginning of the next record. This record becomes the current record. On output, a new record is created and becomes the last record of the file.

For direct access, the file is positioned at the beginning of the record specified by the record specifier. This record becomes the current record.

If the file contains an endfile record, the file must not be positioned after the endfile record prior to data transfer.

9.2.1.3.2. File Position After Data Transfer. If an end-of-file condition exists as a result of reading an endfile record, the file is positioned after the endfile record.

If no error condition or end-of-file condition exists, the file is positioned after the last record read or written and that record becomes the preceding record. A record written on a file connected for sequential access becomes the last record of the file.

If the file is positioned after the endfile record, execution of a data transfer input/output statement is prohibited. However, a REWIND or BACKSPACE statement may be used to reposition the file.

If an error condition exists, the position of the file is indeterminate.

9.2.2. Internal Files. Internal files provide a means of transferring and converting data from internal storage to internal storage.

9.2.2.1. Internal File Properties. An internal file has the following properties:

(1) The file is a character variable.

(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 (6.2.4.2) or the array section (6.2.4.3). Every record of the file has the same length, which is the length of an array element in the array.

(4) A record of the internal file becomes defined by writing the record. If the number of characters written in a record is less than the length of the record, the remaining portion of the record is filled with blanks. If the number of characters to be written is greater than the length of the record, the effect is as though characters equal to the length are written and remaining characters truncated.

(5) A record may be read only if the record is defined.

(6) A record of an internal file may become defined (or undefined) by means other than an output statement. For example, the character variable may become defined by a character assignment statement.

(7) An internal file is always positioned at the beginning of the first record prior to data transfer.

9.2.2.2. Internal File Restrictions. An internal file has the following restrictions:

(1) Reading and writing records must be accomplished only by sequential access formatted input/output statements that do not specify namelist formatting.

(2) An internal file must not be specified in a file connection statement, a file positioning statement, or a file inquiry statement.

9.3. File Connection.

A **unit**, specified by an *io-unit*, provides a means for referring to a file.

R901 *io-unit* **is** *external-file-unit*
or *
or *internal-file-unit*

R902 *external-file-unit* **is** *scalar-int-expr*

R903 *internal-file-unit* **is** *char-variable*

A 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*.

A scalar integer expression that identifies an external file unit must be zero or positive.

The *io-unit* in a file positioning statement, a file connection statement, or a file inquiry statement must be an *external-file-unit*.

The external unit identified by the value of *scalar-int-expr* is the same external unit in all program units of the executable program. In the example:

```
SUBROUTINE A
READ (6) X
.
.
.
SUBROUTINE B
N = 6
REWIND N
```

the value 6 used in both program units identifies the same external unit.

An asterisk identifies a particular processor-dependent external unit that is preconnected for formatted sequential access. This is normally the unit preconnected for the PRINT statement or the unit preconnected for the READ *format* statement.

9.3.1. Unit Existence. At any given time, there is a processor-determined set of external units that are said to exist for an executable program.

All input/output statements may refer to units that exist. The INQUIRE statement and the CLOSE statement also may refer to units that do not exist.

9.3.2. Connection of a File to a Unit. An external unit has a property of being **connected** or not connected. If connected, it refers to a 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 must 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. An example is a preconnected new external file.

A unit must not be connected to more than one file at the same time, and a file must not be connected to more than one unit at the same time. However, means are provided to change the status of 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 executable 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 executable program to the same unit or to a different unit. Note, however, that the only means of referencing a file that has been disconnected is by the appearance of its name in an OPEN or INQUIRE statement. There may be no means of reconnecting an unnamed file once it is disconnected.

An internal unit is always connected to the internal file designated by the *char-variable* that identifies the unit. Note, however, that if the *char-variable* is an allocatable array not currently allocated, an alias object not currently alias associated, or a subobject of either of these, the internal file must not be referenced by an input/output statement.

9.3.3. Preconnection. Preconnection means that the unit is connected to a file at the beginning of execution of the executable program and therefore it may be specified in input/output statements without the prior execution of an OPEN statement.

9.3.4. The OPEN Statement. The **OPEN statement** may be used to connect an existing file to a unit, create a file that is preconnected, create a file and connect it to a unit, or change certain specifiers of a connection between a file and a unit.

An external unit may be connected by an OPEN statement in any program unit of an executable program and, once connected, a reference to it may appear in any program unit of the executable program.

If a unit is connected to a file that exists, execution of an OPEN statement for that unit is permitted. If the FILE= specifier is not included in such an OPEN statement, the file remains connected to the unit.

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 already connected to the unit, only the BLANK=, DELIM=, PAD=, ERR=, and IOSTAT= specifiers may have a value different from the one currently in effect. 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.

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-int-variable*
or ERR= *label*
or FILE= *file-name-expr*
or STATUS= *scalar-char-expr*
or ACCESS= *scalar-char-expr*
or FORM= *scalar-char-expr*
or RECL= *scalar-int-expr*
or BLANK= *scalar-char-expr*
or POSITION= *scalar-char-expr*
or ACTION= *scalar-char-expr*
or DELIM= *scalar-char-expr*
or PAD= *scalar-char-expr*

R906 *file-name-expr* **is** *scalar-char-expr*

Constraint: If the optional characters UNIT= are omitted from the unit specifier, the unit specifier must be the first item in the *connect-spec-list*.

Constraint: Each specifier must not appear more than once in a given *open-stmt*; an *external-file-unit* must be specified.

Constraint: The *label* used in the ERR= specifier must 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 OLD or NEW, the FILE= specifier must be present. If the STATUS= specifier has the value SCRATCH, the FILE= specifier must be absent.

Except for the FILE = specifier, a specifier that requires a *scalar-char-expr* may have a limited list of character values. These values are listed for each such specifier. Any trailing blanks are ignored. If a processor is capable of representing letters in both upper and lower case, the value specified is without regard to case. Some specifiers have a default value if the specifier is omitted.

The IOSTAT= specifier and ERR= specifier are described in 9.4.1.5 and 9.4.1.6, respectively.

An example of an OPEN statement is:

```
OPEN (10, FILE = 'employee.names', ACTION = 'READ', PAD = 'YES')
```

9.3.4.1. FILE= Specifier in the OPEN Statement. The value of the FILE= specifier is the name of the file to be connected to the specified unit. The *file-name-expr* must be a name that is allowed by the processor. If this specifier is omitted and the unit is not connected to a file, it may become connected to a processor-determined file. 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-char-expr* must evaluate to OLD, NEW, SCRATCH, or UNKNOWN. If OLD is specified, the file must exist. If NEW is specified, the file must not exist.

Successful execution of an OPEN statement with NEW specified creates the file and changes the status to OLD. If SCRATCH is specified, the file is created and connected to the specified unit for use by the executable program but is deleted at the execution of a CLOSE statement referring to the same unit or at the termination of the executable program. Note that SCRATCH must 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-char-expr* must evaluate to SEQUENTIAL or DIRECT. The ACCESS= specifier specifies the access method for the connection of the file as being sequential or direct. If this specifier is omitted, the default value is SEQUENTIAL. For an existing file, the specified access method must be included in the set of allowed access methods for the file. For a new file, the processor creates the file with a set of allowed access methods that includes the specified method.

9.3.4.4. FORM= Specifier in the OPEN Statement. The *scalar-char-expr* must evaluate to FORMATTED or UNFORMATTED. The FORM= specifier determines whether the file is being connected for formatted or unformatted input/output. If the FORM= specifier is omitted, 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 must be included in the set of allowed forms for the file. For a new file, the processor creates the file with a set of allowed forms that includes the specified form.

9.3.4.5. RECL= Specifier in the OPEN Statement. The value of the RECL= specifier must be positive. It specifies the length of each record in a file being connected for direct access, or specifies the maximal length of a record in a file being connected for sequential access. If the file is being connected for formatted input/output, the length is the number of characters. If the file is being connected for unformatted input/output, the length is measured in processor-dependent units. For an existing file, the value of the RECL= specifier must be included in the set of allowed record lengths for the file. For a new file, the processor creates the file with a set of allowed record lengths that includes the specified value.

9.3.4.6 BLANK= Specifier in the OPEN Statement. The *scalar-char-expr* must evaluate to NULL or ZERO. The BLANK= specifier is permitted only for a file being connected for formatted input/output. If NULL is specified, all blank characters in numeric formatted input fields on the specified unit are ignored, except that a field of all blanks has a value of zero. If ZERO is specified, all blanks other than leading blanks are treated as zeros. If the BLANK= specifier is omitted, the default value is NULL.

9.3.4.7 POSITION= Specifier in the OPEN Statement. The *scalar-char-expr* must evaluate to ASIS, REWIND, or APPEND. The connection must 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. 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-char-expr* must evaluate to READ, WRITE, or READ/WRITE. READ specifies that the WRITE, PRINT, and ENDFILE statements must not refer to this connection. WRITE specifies that READ statements must not refer to this connection, READ/WRITE permits any I/O statements to refer to this connection. If this specifier is omitted, the default value is READ/WRITE.

9.3.4.9 DELIM= Specifier in the OPEN Statement. The *scalar-char-expr* must evaluate to APOSTROPHE, QUOTE, or NONE. If APOSTROPHE is specified, the apostrophe will be used to delimit character constants written with list-directed or namelist formatting and all internal apostrophes will be doubled. If QUOTE is specified, the quotation mark will be used to delimit character constants written with list-directed or namelist formatting and all internal quotation marks will be doubled. If the value of this specifier is NONE, a character constant when written will not be delimited by apostrophes or quotation marks, nor will 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-char-expr* must evaluate to YES or NO. If YES is specified, a formatted input record is logically padded with blanks when an input list is specified and the format specification requires more data from a record than the record contains. If NO is specified, the input list and the format specification must not require more characters from a record than the record contains. If this specifier is omitted. the default value is YES.

9.3.5 The CLOSE Statement. The **CLOSE statement** is used to terminate the connection of a particular file to a unit.

R907 *close-stmt* **is** CLOSE (*close-spec-list*)

R908 *close-spec* **is** [UNIT=] *external-file-unit*
or IOSTAT= *scalar-int-variable*
or ERR= *label*
or STATUS= *scalar-char-expr*

Constraint: If the optional characters UNIT= are omitted from the unit specifier, the unit specifier must be the first item in the *close-spec-list*.

Constraint: The *label* used in the ERR= specifier must be the statement label of a branch target statement that appears in the same scoping unit as the CLOSE statement.

Constraint: A given specifier must not appear more than once in a given *close-stmt*; the unit specifier must appear.

The IOSTAT= specifier and ERR= specifier are described in Sections 9.4.1.5 and 9.4.1.6, respectively.

A specifier that requires a *scalar-char-expr* may have a limited list of character values. These values are listed for each such specifier. Any trailing blanks are ignored. If a processor is capable of representing letters in both upper and lower case, the value specified is without regard to case. Some specifiers have a default value if the specifier is omitted.

Execution of a CLOSE statement that refers to a unit may occur in any program unit of an executable program and need not occur in the same program unit as the execution of an OPEN statement referring to that unit.

Execution of a CLOSE statement specifying a unit that does not exist or has no file connected to it is permitted and affects no file.

After a unit has been disconnected by execution of a CLOSE statement, it may be connected again within the same executable program, either to the same file or to a different file. After a file has been disconnected by execution of a CLOSE statement, it may be connected again within the same executable program, either to the same unit or to a different unit, provided that the file still exists.

At termination of execution of an executable program for reasons other than an error condition, all units that are connected are closed. Each unit is closed with status KEEP unless the file status prior to termination of execution was SCRATCH, in which case the unit is closed with status DELETE. Note that the effect is as though a CLOSE statement without a STATUS= specifier were executed on each connected unit.

An example of a CLOSE statement is:

```
CLOSE (10, STATUS = 'KEEP')
```

9.3.5.1. STATUS= Specifier in the CLOSE Statement. The *scalar-char-expr* must evaluate to KEEP or DELETE. The STATUS= specifier determines the disposition of the file that is connected to the specified unit. KEEP must not be specified for a file whose status prior to execution of a CLOSE statement is SCRATCH. If KEEP is specified for a file that exists, the file continues to exist after the execution of a CLOSE statement. If KEEP is specified for a file that does not exist, the file will not exist after the execution of a CLOSE statement. If DELETE is specified, the file will not exist after the execution of a CLOSE statement. If this specifier is omitted, the 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 **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*]

Examples of data transfer statements are:

```
   READ (6, *, PROMPT = ' SIZE:') 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 an input prompt string

(5) Optional specification of exception handling

(6) Optional return of counts of values transmitted and values skipped

(7) Optional return of status

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 PROMPT = *scalar-char-expr*
or IOSTAT = *scalar-int-variable*
or ERR = *label*
or END = *label*
or NULLS = *scalar-int-variable*
or VALUES = *scalar-int-variable*

Constraint: An *io-control-spec-list* must contain exactly one *io-unit* and may contain at most one of each of the other specifiers.

Constraint: An END=, a NULLS=, or a PROMPT= specifier must not appear in a *write-stmt* or *print-stmt*.

Constraint: The *label* used in the ERR= or END= specifier must 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* must 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* must 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 must be the first item in the control information list.

Constraint: If the optional characters FMT= are omitted from the format specifier, the format specifier must be the second item in the control information list and the first item must be the unit specifier without the optional characters UNIT=.

Constraint: If the optional characters NML= are omitted from the namelist specifier, the namelist specifier must be the second item in the control information list and the first item must be the unit specifier without the optional characters UNIT=.

Constraint: If the unit specifier specifies an internal file, the *io-control-spec-list* must not contain a REC= specifier or a *namelist-group-name*.

Constraint: A NULLS= specifier may be present only in a list-directed input statement.

Constraint: If a *namelist-group-name* is present, a NULLS = specifier must not appear.

Constraint: If a *namelist-group-name* is present, a VALUES= specifier must not appear.

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 variables specified in IOSTAT=, NULLS=, or VALUES= specifiers, if any, must not be associated with one another, nor 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=, NULLS=, or VALUES= specifier is an array element reference, its subscript values must 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*.

An example of a READ statement is:

```
READ (IOSTAT = IOS, UNIT = 6, FMT = '(10F8.2)', VALUES = NVALS) A, B
```

9.4.1.1. Format Specifier.

R913 *format* **is** *char-expr*
or *label*
or *
or scalar-int-variable

Constraint: The *label* must be the label of a FORMAT statement that appears in the same scoping unit as the statement containing the format specifier.

The *scalar-int-variable* must have been assigned (8.2.4) the statement label of a FORMAT statement that appears in the same scoping unit as the *format*.

The *char-expr* must evaluate to a character object that is a valid format specification (10.1.1). Note that *char-expr* includes a character constant.

If *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** and a REC= specifier must not be present.

An example in which the format is a character expression is:

```
READ (6, FMT = "(" // CHAR_FMT // ")" )  X, Y, Z
```

where CHAR__FMT is a 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 and the file must be connected for direct access. If the control information list contains a REC= specifier, the statement is a **direct access input/output statement** and an END= specifier must not be present; otherwise, it is a **sequential access input/output statement**.

9.4.1.4. Prompt Specifier. For a formatted external READ statement, the scalar character expression specified in the PROMPT= specifier is displayed in a processor-dependent manner, if possible, without line spacing following it. The input statement is then executed. The PROMPT= specifier is not permitted in a WRITE or PRINT statement.

9.4.1.5. 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 nor an end-of-file condition is encountered by the processor,

(2) With a processor-dependent positive integer value if an error condition is encountered, or

(3) With a processor-dependent negative integer value if an end-of-file condition is encountered and no error condition is encountered. Note that this condition may occur only during a sequential input statement.

Consider the example:

```
READ (FMT = "(E8.3)", UNIT=3, IOSTAT = IOSS) X
   !
IF (IOSS < 0) THEN
   !
   ! PERFORM END-OF-FILE PROCESSING ON THE FILE
   ! CONNECTED TO UNIT 3.
   CALL END_PROCESSING
   !
ELSE IF (IOSS > 0) THEN
   !
   ! PERFORM ERROR PROCESSING
   CALL ERROR_PROCESSING
   !
END IF
```

9.4.1.6. Error Branch. If an input/output statement contains an ERR= specifier and the processor encounters an error condition during execution of the statement:

(1) Execution of the input/output statement terminates,

(2) The position of the file specified in the input/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, and

(4) Execution continues with the statement specified in the ERR= specifier. The labeled statement must be in the same scoping unit as the input/output statement.

9.4.1.7. End-of-File Branch. If an input statement contains an END= specifier and the processor encounters an end-of-file condition and encounters no error condition during execution of the statement:

(1) Execution of the READ statement terminates,

(2) If the input statement also contains an IOSTAT= specifier, the variable specified becomes defined with a processor-dependent negative integer value, and

(3) Execution continues with the statement specified in the END= specifier. The labeled statement must be in the same scoping unit as the input/output statement.

In a WRITE statement, the control information list must not contain an END= specifier.

9.4.1.8. Nulls Count. A **null value** is a value that has no effect on the definition status of the corresponding input list item. If the input list item is defined, it retains its previous value; if it is undefined, it remains undefined. A null value must not be used as either the real or imaginary part of a complex constant, but a single null value may represent an entire complex constant.

When an input statement terminates, the variable specified in the NULLS= specifier is defined to be the count of the null values read by the input statement. The value of the variable can be nonzero only for list-directed input.

The NULLS= specifier must not appear in a namelist READ statement.

9.4.1.9. Values Count. When an input/output statement terminates, the variable specified in the VALUES= specifier is defined to be the count of the number of values successfully read or written, with or without editing, by the input/output statement.

Any null values are included in the count of values.

The VALUES= specifier must not appear in a namelist data transfer statement.

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-numeric-expr* , ■
■ *scalar-numeric-expr* [, *scalar-numeric-expr*]

Constraint: The *do-variable* must be scalar of type integer, default real, or double precision real.

Constraint: In an *input-item-list*, an *io-implied-do-object* must be an *input-item*. In an *output-item-list*, an *io-implied-do-object* must be an *output-item*.

An *input-item* must not appear as, nor be associated with, the *do-variable* of any *io-implied-do* that contains the *input-item*.

The *do-variable* of an *io-implied-do* that is contained within another *io-implied-do* must not appear as, nor be associated with, the *do-variable* of the containing *io-implied-do*.

If an array appears as an input/output list item, it is treated as if the elements, if any, were specified in array element order (6.2.4.2). The appearance of a ranged array as a whole array is interpreted as a reference to the elements in its effective range. The name or array section of an assumed-size array must not appear as an input/output list item.

The name of a derived-type object must not appear as an input/output list item if any component ultimately contained within the object is not accessible within the scoping unit containing the input/output statement.

If the name of 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. The values count associated with the derived-type object is that of the objects of intrinsic data type that result from this treatment.

If the name or subobject designator of a derived-type object appears as an input/output list item in an unformatted input/output statement, it is treated as a single value in a processor-dependent form. Note that, in this case, the appearance of a derived-type object as an input/output list item is not equivalent to the list of its components.

The values count associated with a scalar object appearing as an input/output list item of an intrinsic type is always one. For example, if X and Z are scalar variables of type real and complex, respectively, the values count associated to the list X, Z is two, even though three external values may be read or written.

For an implied do, the loop initialization and execution is the same as for a DO construct (8.1.4.4).

Note that a constant, an expression involving operators or function references, or an expression enclosed in parentheses may appear as an output list item but must not appear as an input list item.

An 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 and End-of-File Conditions. The set of input/output error conditions is processor dependent.

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

(1) An endfile record is encountered during the reading of a file connected for sequential access. In this case, the file is positioned after the endfile record.

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

Note that an end-of-file condition can occur at the beginning of an input statement or within a formatted input statement when more than one record is required by the interaction of the input/output list and the format.

If an error condition or an end-of-file condition occurs during execution of an input/output statement, execution of the input/output statement terminates. If an error condition occurs during execution of an input/output statement, the position of the file becomes indeterminate. The VALUES= specifier, if any, is defined with the count of values successfully read or written. On input, any remaining list items are undefined. For any specific error condition, the number of values defined is processor dependent. If the VALUES= specifier is not present, all list items are undefined. Note that for list-directed input, some elements of the input list may not have had their definition status changed due to null values.

When the VALUES= specifier is present, the definition status of any DO variable in the input/output list is determined as follows: If the last successful transfer of a value took place while one or more *io-implied-do*s were active, the DO variables are defined with the values they had at the time the error or end-of-file condition was detected. Any DO variable defined prior to the detection of the error or end-of-file condition in the matching process remains defined. Any remaining *do-variable* in the input/output list is undefined. If the VALUES= specifier is not present, all DO variables in the input list are undefined.

If an error condition occurs during execution of an input/output statement that contains neither an IOSTAT= nor an ERR= specifier, or if an end-of-file condition occurs during execution of a READ statement that contains neither an IOSTAT= specifier nor an END= specifier, execution of the executable program is terminated.

9.4.4. Execution of a Data Transfer Input/Output Statement. The effect of executing a data transfer input/output statement must be as if the following operations were performed in the order specified:

(1) Determine the direction of data transfer

(2) Identify the unit

(3) Establish the format if one is specified

(4) Position the file prior to data transfer (9.2.1.3.1)

(5) Transfer the value of the PROMPT= specifier, if any

(6) Transfer data between the file and the entities specified by the input/output list (if any) or namelist

(7) Position the file after data transfer (9.2.1.3.2)

(8) Cause the variables specified in the IOSTAT=, VALUES=, and NULLS= specifiers, if specified, 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 file unit specifier that identifies an external unit or an internal file. A READ statement that does not contain an input/output control list specifies a particular processor-determined unit, which is the same as the unit identified by * in a READ statement that contains an input/output control list. The PRINT statement specifies some other processor-determined unit, which is the same as the unit identified by * in a WRITE statement. Thus, each data transfer input/output statement identifies an external unit or an internal file.

The unit identified by a data transfer input/output statement must be connected to a file when execution of the statement begins. Note that the file may be preconnected.

9.4.4.3. 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. 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 must not be specified.

9.4.4.4. Data Transfer. Data are transferred between records and entities specified by the input/output list. The list items are processed in the order of the input/output list for all data transfer input/output statements except namelist formatted data transfer input statements. The list items for a namelist formatted data transfer input statement are processed in the order of the entities specified within the input records.

All values needed to determine which entities are specified by an input/output list item are determined at the beginning of the processing of that item.

All values are transmitted to or from the entities specified by a list item prior to the processing of any succeeding list item for all data transfer input/output statements. 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 formatted data transfer 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, must not contain any portion of the established format specification.

If an internal file has been specified, an input/output list item must not be in the file or associated with the file. Note that the file is a character 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 must be defined.

On input, an attempt to read a record of a file connected for direct access that has not previously been written causes all entities specified by the input list to become undefined unless one or more formatted records have been read by this READ statement and VALUES= has been specified.

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 through an unformatted data transfer statement.

On input, the file must be positioned so that the record read is an unformatted record or an endfile record.

On input, the number of values required by the input list must be less than or equal to the number of values in the record.

On input, the type of each value in the record must agree with the type of the corresponding entity in the input list, except that one complex value may correspond to two real list entities or two real values may correspond to one complex list entity. If an entity in the input list is of type character, the length of the character entity must agree with the length of the character value.

On output to a file connected for direct access, the output list must not specify more values than can fit into the record.

On output, if the file is connected for direct access and the values specified by the output list do not fill the record, the remainder of the record is undefined.

If the file is connected for formatted input/output, unformatted data transfer is prohibited.

The unit specified must be an external unit.

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 transmitted through a formatted data transfer statement to or from objects of intrinsic or derived types. In the latter case, the transmission is in the form of values of intrinsic types to or from the components of intrinsic types which ultimately comprise these structured objects.

On input, the file must be positioned so that the record read is a formatted record or an endfile record.

If the file is connected for unformatted input/output, formatted data transfer is prohibited.

The input record is logically padded with blanks to satisfy an input list and format specification that requires more characters than the record contains, unless PAD = NO was specified in the OPEN statement. If PAD = NO was specified, the input list and format specification must not 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 must not specify more characters for a record than have been specified by a RECL= specifier in the OPEN statement.

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 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) All elements of the *input-item-list* or *output-item-list* have been read or written, with or without editing, to or from the specified file.

(2) An error condition is encountered.

(3) An end-of-file condition is encountered.

(4) A slash (/) is encountered as a value separator (10.8, 10.9) in the record being read during list-directed or namelist input.

9.5. File Positioning Statements.

R919 *backspace-stmt* **is** BACKSPACE *external-file-unit*
or BACKSPACE (*position-spec-list*)

R920 *endfile-stmt* **is** ENDFILE *external-file-unit*
or ENDFILE (*position-spec-list*)

R921 *rewind-stmt* **is** REWIND *external-file-unit*
or REWIND (*position-spec-list*)

Constraint: BACKSPACE, ENDFILE, and REWIND apply only to external files.

A file that is not connected for sequential access must not be referred to by a BACKSPACE, an ENDFILE, or a REWIND statement.

R922 *position-spec* **is** [UNIT =] *external-file-unit*
or IOSTAT = *scalar-int-variable*
or ERR = *label*

Constraint: The *label* used in the ERR= specifier must 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 must be the first item in the *position-spec-list*.

Constraint: A *position-spec-list* must contain exactly one *external-file-unit* and may contain at most one of each of the other specifiers.

9.5.1 BACKSPACE Statement. Execution of a BACKSPACE statement causes the file connected to the specified unit to be positioned before the preceding record. If there is no preceding record, the position of the file is not changed. Note that if the preceding record is an endfile record, the file becomes positioned before the endfile record.

Backspacing a file that is connected but does not exist is prohibited.

Backspacing over records written using list-directed or namelist formatting is prohibited.

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. If the file may also be connected for direct access, only those records before the endfile record are considered to have been written. Thus, only those records may be read during subsequent direct access connections to the file.

After execution of an ENDFILE statement, a BACKSPACE or REWIND statement must be used to reposition the file prior to execution of any data transfer input/output statement.

Execution of an ENDFILE statement for a file that is connected but does not exist creates the file.

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 that if the file is already positioned at its initial point, execution of this statement has no effect on the position of the file.

Execution of a REWIND statement for a file that is connected but does not exist is permitted but has no effect.

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-int-variable*) *output-item-list*

Examples of INQUIRE statements are:

```
INQUIRE (IOLENGTH = IOL) A (1:N)
INQUIRE (UNIT = JOAN, OPENED = LOG_01, NAMED = LOG_02, &
```

```
      FORM = CHAR_VAR, IOSTAT = IOS)
```

9.6.1. 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-int-variable*
or ERR = *label*
or EXIST = *scalar-logical-variable*
or OPENED = *scalar-logical-variable*
or NUMBER = *scalar-int-variable*
or NAMED = *scalar-logical-variable*
or NAME = *scalar-char-variable*
or ACCESS = *scalar-char-variable*
or SEQUENTIAL = *scalar-char-variable*
or DIRECT = *scalar-char-variable*
or FORM = *scalar-char-variable*
or FORMATTED = *scalar-char-variable*
or UNFORMATTED = *scalar-char-variable*
or RECL = *scalar-int-variable*
or NEXTREC = *scalar-int-variable*
or BLANK = *scalar-char-variable*
or POSITION = *scalar-char-variable*
or ACTION = *scalar-char-variable*
or DELIM = *scalar-char-variable*
or PAD = *scalar-char-variable*

Constraint: An INQUIRE statement must contain one FILE= specifier or one UNIT= specifier, but not both, and at most one of each of the other specifiers.

Constraint: In the inquire by unit form of the INQUIRE statement, if the optional characters UNIT= are omitted from the unit specifier, the unit specifier must 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).

9.6.1.1. FILE= Specifier in the INQUIRE Statement. The value of *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 *file-name-expr* must be of a form acceptable to the processor as a file name. 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-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-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 *scalar-logical-variable* to be assigned the value true if the specified unit is connected to a file; otherwise, false is assigned.

9.6.1.4. NUMBER= Specifier in the INQUIRE Statement. The *scalar-int-variable* in the NUMBER= specifier is assigned the value of the external unit identifier of the unit that is currently connected to the file. If there is no unit connected to the file, the value −1 is assigned.

9.6.1.5. NAMED= Specifier in the INQUIRE Statement. The *scalar-logical-variable* in the NAMED= specifier is assigned the value true if the file has a name; otherwise, it is assigned the value false.

9.6.1.6. NAME= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the NAME= specifier is assigned the value of the name of the file if the file has a name; otherwise, it becomes undefined. Note that if this specifier appears in an INQUIRE by file statement, its value is not necessarily the same as the name given in the FILE= specifier. For example, the processor may return a file name qualified by a user identification. However, the value returned must be suitable for use as the value of *file-name-expr* in the FILE= specifier in an OPEN statement. If a processor is capable of representing letters in both upper and lower case, the interpretation of case is processor dependent.

9.6.1.7. ACCESS= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the ACCESS= specifier is assigned the value SEQUENTIAL if the file is connected for sequential access, and DIRECT if the file is connected for direct access. If there is no connection, it is assigned the value UNDEFINED.

9.6.1.8. SEQUENTIAL= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the SEQUENTIAL= specifier is assigned the value YES if SEQUENTIAL is included in the set of allowed access methods for the file, NO if SEQUENTIAL is not included in the set of allowed access methods for the file, and UNKNOWN if the processor is unable to determine whether or not SEQUENTIAL is included in the set of allowed access methods for the file.

9.6.1.9. DIRECT= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the DIRECT= specifier is assigned the value YES if DIRECT is included in the set of allowed access methods for the file, NO if DIRECT is not included in the set of allowed access methods for the file, and UNKNOWN if the processor is unable to determine whether or not DIRECT is included in the set of allowed access methods for the file.

9.6.1.10. FORM= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the FORM= specifier is assigned the value FORMATTED if the file is connected for formatted input/output, and is assigned the value UNFORMATTED if the file is connected for unformatted input/output. If there is no connection, it is assigned the value UNDEFINED.

9.6.1.11. FORMATTED= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the FORMATTED= specifier is assigned the value YES if FORMATTED is included in the set of allowed forms for the file, NO if FORMATTED is not included in the set of allowed forms for the file, and UNKNOWN if the processor is unable to determine whether or not FORMATTED is included in the set of allowed forms for the file.

9.6.1.12. UNFORMATTED= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the UNFORMATTED= specifier is assigned the value YES if UNFORMATTED is included in the set of allowed forms for the file, NO if UNFORMATTED is not included in the set of allowed forms for the file, and UNKNOWN if the processor is unable to determine whether or not UNFORMATTED is included in the set of allowed forms for the file.

9.6.1.13 RECL= Specifier in the INQUIRE Statement. The *scalar-int-variable* in the RECL= specifier is assigned the value of the maximal record length of the file. If the file is connected for formatted input/output, the length is the number of characters. If the file is connected for unformatted input/output, the length is measured in processor-defined units. If the file does not exist, *scalar-int-variable* becomes undefined.

9.6.1.14 NEXTREC= Specifier in the INQUIRE Statement. The *scalar-int-variable* in the NEXTREC= specifier is assigned the value $n + 1$, where n is the record number of the last record read or written on the file connected for direct access. If the file is connected but no records have been read or written since the connection, *scalar-int-variable* is assigned the value 1. If the file is not connected for direct access or if the position of the file is indeterminate because of a previous error condition, *scalar-int-variable* becomes undefined.

9.6.1.15 BLANK= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the BLANK= specifier is assigned the value NULL if null blank control is in effect for the file connected for formatted input/output, and is assigned the value ZERO if zero blank control is in effect for the file connected for formatted input/output. If there is no connection, or if the connection is not for formatted input/output, *scalar-char-variable* is assigned the value UNDEFINED.

9.6.1.16 POSITION= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the POSITION= specifier is assigned the value REWIND if the file is connected by an OPEN statement for positioning at its initial point, APPEND if the file is connected for positioning at its terminal point, and ASIS if the file is connected without changing its position. If there is no connection, *scalar-char-variable* is assigned the value UNDEFINED. If the file has been repositioned since the connection, *scalar-char-variable* is assigned the value UNDEFINED.

9.6.1.17 ACTION= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the ACTION= specifier is assigned the value READ if the file is connected for input only, WRITE if the file is connected for output only, and READ/WRITE if it is connected for both input and output. If there is no connection, *scalar-char-variable* is assigned the value UNDEFINED.

9.6.1.18 DELIM= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the DELIM= specifier is assigned the value APOSTROPHE if the apostrophe is to be used to delimit character data written by list-directed or 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, *scalar-char-variable* is assigned the value UNDEFINED.

9.6.1.19 PAD= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the PAD= specifier is assigned the value NO if the connection of the file to the unit included the PAD= specifier and its value was NO. Otherwise, *scalar-char-variable* is assigned the value YES.

9.6.1.20 IOLENGTH= Specifier in the INQUIRE Statement. The *scalar-int-variable* in the IOLENGTH= specifier is assigned the processor-dependent value that would result from the use of the output list in an unformatted output statement. Any DO variables have the scope of the implied-DO list, as in the DATA statement. The value must be suitable as a RECL= specifier in an OPEN statement that connects a file for unformatted direct access when there are input/output statements with the same input/output list.

9.6.1.21 Restrictions on Inquiry Specifiers. A variable that may become defined or undefined as a result of its use in a specifier in an INQUIRE statement, or any associated entity, must not appear in another specifier in the same INQUIRE statement.

The *inquire-spec-list* in an INQUIRE by file statement must contain exactly one FILE= specifier and must not contain a UNIT= specifier.

The *inquire-spec-list* in an INQUIRE by unit statement must contain exactly one UNIT= specifier and must not contain a FILE= specifier. The unit specified need not exist or be connected to a file. If it is connected to a file, the inquiry is being made about the connection and about the file connected.

9.7. Restrictions on Function References and List Items. A function reference must not appear in an expression anywhere in an input/output statement if such a reference causes another input/output statement to be executed. Note that restrictions in the evaluation of expressions (7.1.7) prohibit certain side effects.

9.8. Restriction on Input/Output Statements. If a unit, or a file connected to a unit, does not have all of the properties required for the execution of certain input/output statements, those statements must not refer to the unit.

10. INPUT/OUTPUT EDITING

A format used in conjunction with an input/output statement provides information that directs the editing between the internal representation of data and the character strings of a record or a sequence of records in a file.

A format specifier (9.4.1.1) in an input/output statement may refer to a FORMAT statement or to a character expression that contains a format specification. A format specification provides explicit editing information. The format specifier also may be an asterisk (*) which indicates list-directed formatting (10.8), or a *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) As the value of a character expression

10.1.1. FORMAT Statement.

R1001 *format-stmt* **is** FORMAT *format-specification*

R1002 *format-specification* **is** ([*format-item-list*])

Constraint: The *format-stmt* must be labeled.

Constraint: The comma used to separate *format-items* in a *format-item-list* may be omitted as follows:

(1) Between a P edit descriptor and an immediately following F, E, EN, D, or G edit descriptor (10.6.5)

(2) Before or after a slash edit descriptor when the optional repeat specification is not present (10.6.2)

(3) Before or after a colon edit descriptor (10.6.3)

Note that, for source form purposes, the format specification is considered to be a form of character context (3.3).

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 must evaluate to a character string whose value constitutes a valid format specification. Note that the format specification begins with a left parenthesis and ends with a right parenthesis.

All character positions up to and including the final right parenthesis of the format specification must be defined at the time the input/output statement is executed, and must not become redefined or undefined during the execution of the statement. Character positions, if any, following the right parenthesis that ends the format specification need not be defined and may contain any character data with no effect on the format specification.

If the format specifier identifies a character array entity, 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 refers to a character array element, the format specification must be contained entirely within that array element.

10.2. Form of a Format Item List.

R1003 *format-item* **is** [*r*] *data-edit-desc*
or *control-edit-desc*
or *char-string-edit-desc*
or [*r*] (*format-item-list*)

R1004 *r* **is** *int-literal-constant*

Constraint: *r* must be positive. It is called a **repeat specification**.

Blank characters may precede the initial left parenthesis of the format specification. Additional blank characters may appear at any point within the format specification, with no effect on the format specification, except within a character string edit descriptor (10.7.1, 10.7.2).

10.2.1. Edit Descriptors. An **edit descriptor** is used to specify the form of a record and to direct the editing between the characters in a record and internal representations of data. The internal representation of a datum corresponds to the internal representation of a constant of the corresponding type.

An edit descriptor is either a **data edit descriptor**, a **control edit descriptor**, or a **character string edit descriptor**.

R1005 *data-edit-desc* **is** I *w* [. *m*]
or F *w* . *d*
or E *w* . *d* [E *e*]
or EN *w* . *d* [E *e*]
or G *w* . *d* [E *e*]
or 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: *w* and *e* must be positive and *d* and *m* must be zero or positive.

The value of *m*, *d*, and *e* may be restricted further by the value of *w*. I, F, E, EN, 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*

R1012 *position-edit-desc* **is** T *n*
or TL *n*
or TR *n*
or *n* X

R1013 *n* **is** *int-literal-constant*

Constraint: *n* must be positive.

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*
or *c* H *character* [*character*]...

R1017 *c* **is** *int-literal-constant*

Constraint: *c* must be positive.

Each character in a character string edit descriptor must be one of the characters capable of representation by the processor.

The character string edit descriptors provide constant data to be output, and are not valid for input.

Within a character constant, appearances of the delimiter character itself, apostrophe or quote, must be as consecutive pairs without intervening blanks. Each such pair represents a single occurrence of the delimiter character.

In the H edit descriptor, *c* specifies the number of characters following the H that comprise the descriptor.

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 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**. Each action of format control depends on information jointly provided by:

(1) The next edit descriptor contained in the format specification, and

(2) The next effective item in the input/output list, if one exists. Zero-sized arrays, zero-sized array sections, zero-length character entities, and implied-DO lists with iteration counts of zero are ignored in determining the next effective item.

If an input/output list specifies at least one list item, at least one data edit descriptor must exist in the format specification. Note that an empty format specification of the form ( ) may be used only if the input/output list is empty or each item is of zero size or length; in this case, one input record is skipped or one output record containing no characters is written. Except for a format item preceded by a repeat specification *r*, a format specification is interpreted from left to right.

A format item preceded by a repeat specification is processed as a list of *r* items, each identical to the format item but without the repeat specification and separated by commas. Note that an omitted repeat specification is treated in the same way as a repeat specification whose value is one.

To each data edit descriptor interpreted in a format specification, there corresponds one effective item specified by the input/output list (9.4.2), except that an input/output list item of type complex requires the interpretation of two F, E, EN, D, or G edit descriptors. For each control edit descriptor or character edit descriptor, there is no corresponding item specified by the input/output list, and format control communicates information directly with the record.

Whenever format control encounters a data edit descriptor in a format specification, it determines whether there is a corresponding effective item specified by the input/output list. If there is such an item, it transmits appropriately edited information between the item and the record, and then format control proceeds. If there is no such item, format control terminates.

If format control encounters a colon edit descriptor in a format specification and another effective input/output list item is not specified, format control terminates.

If format control encounters the rightmost parenthesis of a complete format specification and another effective input/output list item is not specified, format control terminates. However, if another effective input/output list item is specified, the file is positioned at the beginning of the next record and format control then reverts to the beginning of the format item terminated by the last preceding right parenthesis. If there is no such preceding right parenthesis, format control reverts to the first left parenthesis of the format specification. If such reversion occurs, the reused portion of the format specification must contain at least one data edit descriptor. If format control reverts to a parenthesis that is preceded by a repeat specification, the repeat specification is reused. Reversion of format control, of itself, has no effect on the scale factor (10.6.5.1), the sign control edit descriptors (10.6.4), or the blank interpretation edit descriptors (10.6.6).

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, X, or slash edit descriptor is processed, the file is positioned as described in 10.6.1.

If format control reverts as described in 10.3, the file is positioned in a manner identical to the way it is positioned when a slash edit descriptor is processed (10.6.2).

During a read operation, any unprocessed characters of the record are skipped whenever the next record is read.

10.5. Data Edit Descriptors. Data edit descriptors cause the conversion of data to or from its internal representation. On input, the specified variable becomes defined unless there is an error or end-of-file condition and no VALUES= specifier is present. On output, the specified expression is evaluated.

10.5.1. Numeric Editing. The I, F, E, EN, D, and G edit descriptors are used to specify the input/output of integer, real, and complex data. The following general rules apply:

(1) On input, leading blanks are not significant. The interpretation of blanks, other than leading blanks, is determined by a combination of any BLANK= specifier (9.3.4.6) and any BN or BZ blank control that is currently in effect for the unit (10.6.6). Plus signs may be omitted. A field containing only blanks is considered to be zero.

(2) On input, with F, E, EN, D, and G editing, a decimal point appearing in the input field overrides the portion of an edit descriptor that specifies the decimal point location. The input field may have more digits than the processor uses to approximate the value of the datum.

(3) On output, the representation of a positive or zero internal value in the field may be prefixed with a plus, as controlled by the S, SP, and SS edit descriptors or the processor. The representation of a negative internal value in the field must be prefixed with a minus. However, the processor must not produce a negative signed zero in a formatted output record.

(4) On output, the representation is right-justified in the field. If the number of characters produced by the editing is smaller than the field width, leading blanks will be inserted in the field.

(5) On output, if the number of characters produced exceeds the field width or if an exponent exceeds its specified length using the E*w.d*E*e*, EN*w.d*E*e*, or G*w.d*E*e* edit descriptor, the processor must fill the entire field of width *w* with asterisks. However, the processor must not produce asterisks if the field width is not exceeded when optional characters are omitted. Note that when an SP edit descriptor is in effect, a plus is not optional.

10.5.1.1. Integer Editing. The I*w* and I*w.m* edit descriptors indicate that the field to be edited occupies *w* positions. The specified input/output list item must be of type integer.

On input, an I*w.m* edit descriptor is treated identically to an I*w* edit descriptor.

In the input field, the character string must be in the form of an optionally signed integer constant, except for the interpretation of blanks.

The output field for the I*w* edit descriptor consists of zero or more leading blanks followed by a minus if the value of the internal datum is negative, or an optional plus otherwise, followed by the magnitude of the internal value in the form of an unsigned integer constant without leading zeros. Note that an integer constant always consists of at least one digit.

The output field for the I*w.m* edit descriptor is the same as for the I*w* edit descriptor, except that the unsigned integer constant consists of at least *m* digits and, if necessary, has leading zeros. The value of *m* must not exceed the value of *w*. If *m* is zero and the value of the internal datum is zero, the output field consists of only blank characters, regardless of the sign control in effect.

10.5.1.2. Real and Complex Editing. The F, E, EN, D, and G edit descriptors specify the editing of real and complex data. An input/output list item corresponding to an F, E, EN, D, or G edit descriptor must be real or complex.

10.5.1.2.1. F Editing. The F*w.d* edit descriptor indicates that the field occupies *w* positions, the fractional part of which consists of *d* digits.

The input field consists of an optional sign, followed by a string of digits optionally containing a decimal point, including any blanks interpreted as zeros. 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) Explicitly signed integer constant

(2) E followed by zero or more blanks, followed by an optionally signed integer constant

(3) D followed by zero or more blanks, followed by an optionally signed integer constant

An exponent containing a D is processed identically to an exponent containing an E.

Note that 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 must appear if there would otherwise be no digits in the output field.

10.5.1.2.2. E and D Editing. The E*w.d*, D*w.d*, and E*w.d*E*e* edit descriptors indicate that the external field occupies *w* positions, the fractional part of which consists of *d* digits, unless a scale factor greater than one is in effect, and the exponent part consists of *e* digits. The *e* has no effect on input 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:

$$[\ \pm\ ]\ [0]\ .\ x_1 x_2 \cdots x_d\ exp$$

where:

$\pm$ signifies a plus or a minus.

$x_1 x_2 \cdots x_d$ are the *d* most significant digits of the datum value after rounding.

exp is a decimal exponent having one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
E*w.d*	$\lvert exp \rvert \le 99$	$E \pm z_1 z_2$ *or* $\pm 0 z_1 z_2$
	$99 < \lvert exp \rvert \le 999$	$\pm z_1 z_2 z_3$
E*w.d*E*e*	$\lvert exp \rvert \le 10^e - 1$	$E \pm z_1 z_2 \cdots z_e$
D*w.d*	$\lvert exp \rvert \le 99$	$D \pm z_1 z_2$ or $E \pm z_1 z_2$ or $\pm 0 z_1 z_2$
	$99 < \lvert exp \rvert \le 999$	$\pm z_1 z_2 z_3$

where each *z* is a digit. The sign in the exponent is required. A plus sign must be used if the exponent value is zero. The forms E*w.d* and *Dw.d* must 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 mantissa is greater than or equal to one and less than 1000, except when the output value is zero. The scale factor has no effect on output.

The forms of the edit descriptor are EN*w.d* and EN*w.d*E*e* indicating that the external field occupies *w* positions, the fractional part of which consists of *d* digits and the exponent part consists of *e* digits.

The form and interpretation of the input field is the same as for F editing (10.5.1.2.1).

The form of the output field is:

$$[\ \pm\ ]\ yyy.x_1x_2\cdots x_d\ exp$$

where:

$\pm$ signifies a plus or a minus.

yyy are the 1 to 3 decimal digits representative of the most significant digits of the value of the datum after rounding (*yyy* is an integer such that $1 \le yyy \le 999$ or $yyy = 0$).

$x_1x_2\cdots x_d$ are the *d* next most significant digits of the value of the datum after rounding.

exp is a decimal exponent, divisible by three, of one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
EN*w.d*	$\lvert exp \rvert \le 99$	$E\pm z_1z_2$ *or* $\pm 0z_1z_2$
	$99 < \lvert exp \rvert \le 999$	$\pm z_1z_2z_3$
EN*w.d*E*e*	$\lvert exp \rvert \le 10^e - 1$	$E\pm z_1z_2\cdots z_e$

where each *z* is a digit.

The sign in the exponent is required. A plus sign must be used if the exponent value is zero. The form EN*w.d* must not be used if $|exp| > 999$.

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. G Editing. The G*w.d* and G*w.d*E*e* edit descriptors indicate that the external field occupies *w* positions, the fractional part of which consists of a maximum 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 method of representation in the output field depends on the magnitude of the datum being edited. Let *N* be the magnitude of the internal datum. If $0 < N < 0.1$ or $N \ge 10^d$, G*w.d* output editing is the same as *k*PE*w.d* output editing and G*w.d*E*e* output editing is the same as *k*PE*w.d*E*e* output editing, where *k* is the scale factor (10.6.5.1) currently in effect. If $0.1 \le N < 10^d$ or *N* is identically 0, the scale factor has no effect, and the value of *N* determines the editing as follows:

Magnitude of Datum	Equivalent Conversion
$N = 0$	F($w - n$).($d - 1$), n('b')
$0.1 \le N < 1$	F($w - n$).d, n('b')
$1 \le N < 10$	F($w - n$).($d - 1$), n('b')
.	.
.	.
.	.
$10^{d-2} \le N < 10^{d-1}$	F($w - n$).1, n('b')
$10^{d-1} \le N < 10^{d}$	F($w - n$).0, n('b')

where *b* is a blank. *n* is 4 for G*w.d* and *e* + 2 for G*w.d*E*e*.

Note that the scale factor has no effect unless the magnitude of the datum to be edited is outside of the range that permits effective use of F editing.

10.5.1.2.5. Complex Editing. A complex datum consists of a pair of separate real data; therefore, the editing is specified by two F, E, EN, D, or G edit descriptors. The first of the edit descriptors specifies the real part; the second specifies the imaginary part. The two edit descriptors may be different. Control and character string edit descriptors may be processed between the two successive F, E, EN, D, or G edit descriptors.

10.5.2. Logical Editing. The L*w* edit descriptor indicates that the field occupies *w* positions. The specified input/output list item must be of type logical.

The input field consists of optional blanks, optionally followed by a decimal point, followed by a T for true or F for false. The T or F may be followed by additional characters in the field. Note that the logical constants .TRUE. and .FALSE. are acceptable input forms.

The output field consists of *w* − *1* blanks followed by a T or F, depending on whether the value of the internal datum is true or false, respectively.

10.5.3. Character Editing. The A[*w*] edit descriptor is used with an input/output list item of type character.

If a field width *w* is specified with the A edit descriptor, the field consists of *w* characters. If a field width *w* is not specified with the A edit descriptor, the number of characters in the field is the length of the character input/output list item.

Let *len* be the length of the input/output list item. If the specified field width *w* for A input is greater than or equal to *len*, the rightmost *len* characters will be taken from the input field. If the specified field width *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.

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.

The position specified by a T edit descriptor may be in either direction from the current position. On input, this allows portions of a record to be processed more than once, possibly with different editing.

The position specified by an X edit descriptor is forward from the current position. On input, a position beyond the last character of the record may be specified if no characters are transmitted from such positions. Note that an *n*X edit descriptor has the same effect as a TR*n* edit descriptor.

On output, a T, TL, TR, or X edit descriptor does not by itself cause characters to be transmitted and therefore does not by itself affect the length of the record. If characters are transmitted to positions at or after the position specified by a T, TL, TR, or X edit descriptor, positions skipped and not previously filled are filled with blanks. The result is as if the entire record were initially filled with blanks.

On output, a character in the record may be replaced. However, a T, TL, TR, or X edit descriptor never directly causes a character already placed in the record to be replaced. Such edit descriptors may result in positioning such that subsequent editing causes a replacement.

10.6.1.1. T, TL, and TR Editing. The T*n* edit descriptor indicates that the transmission of the next character to or from a record is to occur at the *n*th character position.

The TL*n* edit descriptor indicates that the transmission of the next character to or from the record is to occur at the character position *n* characters backward from the current position. However, if the current position is less than or equal to position *n*, the TL*n* edit descriptor indicates that the transmission of the next character to or from the record is to occur at position one of the current record.

The TR*n* edit descriptor indicates that the transmission of the next character to or from the record is to occur at the character position *n* characters forward from the current position.

Note that *n* must be specified and must be greater than zero.

10.6.1.2. X Editing. The *n*X edit descriptor indicates that the transmission of the next character to or from a record is to occur at the position *n* characters forward from the current position. Note that the *n* must be specified and must be greater than zero.

10.6.2. Slash Editing. The slash edit descriptor indicates the end of data transfer on the current record.

On input from a file connected for sequential access, the remaining portion of the current record is skipped and the file is positioned at the beginning of the next record. This record becomes the current record. On output to a file connected for sequential access, a new record is created and becomes the last and current record of the file.

Note that a record that contains no characters may be written on output. If the file is an internal file or a file connected for direct access, the record is filled with blank characters. Note also that an entire record may be skipped on input. The repeat specification is optional on the slash edit descriptor. If it is not specified, the default value is one.

For a file connected for direct access, the record number is increased by one and the file is positioned at the beginning of the record that has that record number. This record becomes the current record.

10.6.3. Colon Editing. The colon edit descriptor terminates format control if there are no more effective items in the input/output list (9.4.2). The colon edit descriptor has no effect if there are more effective items in the input/output list.

10.6.4. S, SP, and SS Editing. The S, SP, and SS edit descriptors may be used to control optional plus characters in numeric output fields. At the beginning of execution of each formatted output statement, the processor has the option of producing a plus in numeric output fields. If an SP edit descriptor is encountered in a format specification, the processor must produce a plus in any subsequent position that normally contains an optional plus. If an SS edit descriptor is encountered, the processor must not produce a plus in any subsequent

position that normally contains an optional plus. If an S edit descriptor is encountered, the option of producing the plus is restored to the processor.

The S, SP, and SS edit descriptors affect only I, F, E, EN, D, and G editing during the execution of an output statement. The S, SP, and SS edit descriptors have no effect during the execution of an input statement.

10.6.5. P Editing. The *k*P edit descriptor sets the value of the scale factor to *k*. The scale factor may affect the editing of numeric quantities.

10.6.5.1. Scale Factor. The value of the scale factor is zero at the beginning of execution of each input/output statement. It applies to all subsequently interpreted F, E, EN, D, and G edit descriptors until another P edit descriptor is encountered, and then a new scale factor is established. Note that reversion of format control (10.3) does not affect the established scale factor.

The scale factor *k* affects the appropriate editing in the following manner:

(1) On input, with F, E, EN, D, and G editing (provided that no exponent exists in the field) and F output editing, the scale factor effect is that the externally represented number equals the internally represented number multiplied by 10^k.

(2) On input, with F, E, EN, D, and G editing, the scale factor has no effect if there is an exponent in the field.

(3) On output, with E and D editing, the significant (4.3.1.2) part of the quantity to be produced is multiplied by 10^k and the exponent is reduced by *k*.

(4) On output, with G editing, the effect of the scale factor is suspended unless the magnitude of the datum to be edited is outside the range that permits the use of F editing. If the use of E editing is required, the scale factor has the same effect as with E output editing.

(5) On output, with EN editing, the scale factor has no effect.

10.6.6. BN and BZ Editing. The BN and BZ edit descriptors may be used to specify the interpretation of blanks, other than leading blanks, in numeric input fields. At the beginning of execution of each formatted input statement, nonleading blank characters are interpreted as zeros or are ignored, depending on the value of the BLANK= specifier (9.3.4.6) currently in effect for the unit. If a BN edit descriptor is encountered in a format specification, all nonleading blank characters in succeeding numeric input fields are ignored. The effect of ignoring blanks is to treat the input field as if blanks had been removed, the remaining portion of the field right-justified, and the blanks replaced as leading blanks. However, a field containing only blanks has the value zero. If a BZ edit descriptor is encountered in a format specification, all nonleading blank characters in succeeding numeric input fields are treated as zeros.

The BN and BZ edit descriptors affect only I, F, E, EN, D, and G editing during execution of an input statement. They have no effect during execution of an output statement.

10.7. Character String Edit Descriptors. A character string edit descriptor must not be used on input.

10.7.1. Character Constant Edit Descriptor. The character constant edit descriptor causes characters to be written from the enclosed characters of the edit descriptor itself, including blanks. Note that a delimiter is either an apostrophe or quote.

For a character constant edit descriptor, the width of the field is the number of characters contained in, but not including, the delimiting characters. Within the field, two consecutive delimiting characters are counted as a single character.

10.7.2. H Editing. The *c*H edit descriptor causes character information to be written from the next *c* characters (including blanks) following the H of the *c*H edit descriptor in the *format-item-list* itself. If a *c*H edit descriptor occurs within a character constant delimited by apostrophes and the H edit descriptor includes an apostrophe, the apostrophe must be represented by two consecutive apostrophes which are counted as one character in specifying *c*. If a *c*H edit descriptor occurs within a character constant delimited by quotes and the H edit descriptor includes a quote, the quote must be represented by two consecutive quotes which are counted as one character in specifying *c*.

10.8. List-Directed Formatting.

The characters in one or more list-directed records constitute a sequence of values and value separators. The end of a record has the same effect as a blank character, unless it is within a character constant. Any sequence of two or more consecutive blanks is treated as a single blank, unless it is within a character constant.

Each value is either a null value or one of the forms:

c
*r***c*
*r**

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

A **value separator** is one of the following:

(1) A comma optionally preceded by one or more contiguous blanks and optionally followed by one or more contiguous blanks

(2) A slash optionally preceded by one or more contiguous blanks and optionally followed by one or more contiguous blanks

(3) One or more contiguous blanks between two nonblank values or following the last nonblank value, where a nonblank value is a constant, an *r***c* form, or an *r** form.

10.8.1. List-Directed Input. Input forms acceptable to edit descriptors for a given type are acceptable for list-directed formatting, except as noted below. The form of the input value must be acceptable for the type of the input list item. Blanks are never used as zeros, and embedded blanks are not permitted in constants, except within character constants and complex constants as specified below. Note that the end of a record has the effect of a blank, except when it appears within a character constant.

When the corresponding input list item is of type real, the input form is that of a numeric input field. A numeric input field is a field suitable for F editing (10.5.1.2.1) that is assumed to have no fractional digits unless a decimal point appears within the field.

When the corresponding list item is of type complex, the input form consists of a left parenthesis followed by an ordered pair of numeric input fields separated by a comma, and followed by a right parenthesis. The first numeric input field is the real part of the complex constant and the second is the imaginary part. Each of the numeric input fields may be preceded or followed by blanks. The end of a record may occur between the real part and the comma or between the comma and the imaginary part.

When the corresponding list item is of type logical, the input form must not include slashes, blanks, or commas among the optional characters permitted for L editing.

When the corresponding list item is of type character, the input form consists of a character constant. Character constants may be continued from the end of one record to the beginning of the next record, but the end of record must not occur between a doubled apostrophe in an apostrophe-delimited constant, nor between a doubled quote in a quote-delimited constant. The end of the record does not cause a blank or any other character to become part of the

constant. The constant may be continued on as many records as needed. The characters blank, comma, and slash may appear in character constants.

If the corresponding input list item is of type character and:

(1) The character constant does not contain the characters blank, comma, or slash, and

(2) The datum does not cross a record boundary, and

(3) The first nonblank character is not a quotation mark or an apostrophe, and

(4) The leading characters are not numeric followed by an asterisk,

the delimiting apostrophes or quotation marks are not required. If the delimiters are omitted, the character constant is terminated by the first blank, comma, slash character, or end of record and apostrophes and quotation marks within the datum are not to be doubled.

Let *len* be the length of the list item, and let *w* be the length of the character constant. If *len* is less than or equal to *w*, the leftmost *len* characters of the constant are transmitted to the list item. If *len* is greater than *w*, the constant is transmitted to the leftmost *w* characters of the list item and the remaining *len* – *w* characters of the list item are filled with blanks. Note that the effect is as though the constant were assigned to the list item in a character assignment statement (7.5.1.4).

10.8.1.1. Null Values. A null value is specified by having no characters between successive value separators, no characters preceding the first value separator in the first record read by each execution of a list-directed input statement, or the *r*∗ form. Note that the end of a record following any other separator, with or without separating blanks, does not specify a null value. A null value has no effect on the definition status of the corresponding input list item.

A slash encountered as a value separator during execution of a list-directed input statement causes termination of execution of that input statement after the assignment of the previous value. If there are additional items in the input list, the effect is as if null values had been supplied for them. Any DO variable in the input list is defined as though enough null values had been supplied for any remaining input list items.

Note that all blanks in a list-directed input record are considered to be part of some value separator except for the following:

(1) Blanks embedded in a character constant

(2) Embedded blanks surrounding the real or imaginary part of a complex constant

(3) Leading blanks in the first record read by each execution of a list-directed input statement, unless immediately followed by a slash or comma

10.8.2. List-Directed Output. The form of the values produced is the same as that required for input, except as noted otherwise. With the exception of nondelimited character constants, the values are separated by (1) one or more blanks or (2) a comma optionally preceded by one or more blanks and optionally followed by one or more blanks.

The processor may begin new records as necessary, but, except for complex constants and character constants, the end of a record must not occur within a constant and blanks must not appear within a constant.

Logical output constants are T for the value true and F for the value false.

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 cases involved.

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

Character constants produced for a file opened without a DELIM= specifier (9.3.4.9) or with a DELIM= specifier with a value of NONE:

(1) Are not delimited by apostrophes or quotation marks,

(2) Are not preceded or followed by a value separator,

(3) Have each internal apostrophe or quotation mark represented externally by one apostrophe or quotation mark, and

(4) Have a blank character inserted by the processor for carriage control at the beginning of any record that begins with the continuation of a character constant from the preceding record.

Character constants produced for a file opened with a DELIM= specifier with a value of QUOTE are delimited by quotes, are preceded and followed by a value separator, and have each internal quote represented on the external medium by two quotes.

Character constants produced for a file opened with a DELIM= specifier with a value of APOSTROPHE are delimited by apostrophes, are preceded and followed by a value separator, and have each internal apostrophe represented on the external medium by two apostrophes.

If two or more successive values in an output record have identical values, the processor has the option of producing a repeated constant of the form *r***c* instead of the sequence of identical values.

Slashes, as value separators, and null values are not produced by list-directed formatting.

Except for continuation of delimited character constants, each output record begins with a blank character to provide carriage control when the record is printed.

10.9. Namelist Formatting. The characters in one or more namelist records constitute a sequence of **name-value subsequences**, each of which consists of a name followed by an equals and followed by one or more values and value separators. The equals may optionally be preceded or followed by zero, one, or more contiguous blanks. The end of a record has the same effect as a blank character, unless it is within a character constant. Any sequence of two or more consecutive blanks is treated as a single blank, unless it is within a character constant.

The name may be any name in the *namelist-group-object-list* (5.4).

Each value is either a null value or one of the forms:

c
*r***c*
*r**

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

A value separator for namelist formatting is the same as for list-directed (10.8) except that a value separator containing a slash must not immediately precede a value.

10.9.1. Namelist Input. Input for a namelist input statement consists of:

(1) Optional blanks,

(2) The character & followed immediately by the same *namelist-group-name* specified in the namelist input 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 statement.

In each name-value subsequence, the name must be the name of a namelist group object list item with an optional qualification.

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 must correspond to a specific namelist group object name. Subscripts, strides, and substring range expressions used to qualify group object names must be optionally signed integer literal constants. If a namelist group object name is the name of an array, the name in the input record corresponding to it may be either the array name or the name of an element or section of that array, indicated by qualifying the array name with integer constant subscripts, starting points, and ending points. 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 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 must not contain embedded blanks.

10.9.1.2. Acceptable Namelist Input Values. The datum *c* 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 type logical. The form of the input value must 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 the name of a scalar variable of an intrinsic type, the equals must 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.

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 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 must 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. The number of values following the equals must 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. For example, if the name in the input record is the name of an integer array of effective 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.

10.9.1.3. Namelist Group Object List Items. When the corresponding 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 corresponding list item is of type complex, the input form of the input value consists of a left parenthesis followed by an ordered pair of numeric input fields separated by a comma and followed by a right parenthesis. The first numeric input field is the real part of the complex constant and the second part is the imaginary part. Each of the numeric input fields may be preceded or followed by blanks. The end of a record may occur between the real part and the comma or between the comma and the imaginary part.

When the corresponding list item is of type logical, the input form of the input value must not include slashes, blanks, or commas among the optional characters permitted for L editing (10.5.2).

When the corresponding list item is of type character, the input form of the input value consists of a string of characters enclosed in apostrophes or quotation marks. Each apostrophe within a character constant delimited by apostrophes must be represented by two consecutive apostrophes without an intervening blank or end of record. Each quotation mark within a character constant delimited by quotation marks must be represented by two consecutive quotation marks without an intervening blank or end of record. Character constants may be continued from the end of one record to the beginning of the next record. The end of the record does not cause a blank or any other character to become part of the constant. The constant may be continued on as many records as needed. The characters blank, comma, and slash may appear in character constants.

Let *len* be the length of the list item, and let *w* be the length of the character constant. If *len* is less than or equal to *w*, the leftmost *len* characters of the constant are transmitted to the list item. If *len* is greater than *w*, the constant is transmitted to the leftmost *w* characters of the list item and the remaining *len* – *w* characters of the list item are filled with blanks. Note that the effect is as though the constant were assigned to the list item in a character assignment statement (7.5.1.4).

If the corresponding list item is of type character and:

(1) The character constant does not contain the value separators blank, comma, or slash,

(2) The character constant does not cross a record boundary,

(3) The first nonblank character is not a quotation mark or an apostrophe, and

(4) The leading characters are not numeric followed by an asterisk,

then the enclosing apostrophes or quotation marks are not required and apostrophes or quotation marks within the character constant are not to be doubled.

10.9.1.4. 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 must not be used as either the real or imaginary part of a complex constant, but a single null value may represent an entire complex constant.

Note that the end of a record following a value separator, with or without intervening blanks, does not specify a null value.

10.9.1.5. 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.

10.9.2. Namelist Output. The form of the output produced is the same as that required for input, except for the forms of real and logical constants. If the processor is capable of representing letters in both upper and lower case, the name in the output is in upper case. With the exception of nondelimited character constants, the values are separated by (1) one or more blanks or (2) a comma optionally preceded by one or more blanks and optionally followed by one or more blanks.

The processor may begin new records as necessary. However, except for complex constants and character constants, the end of a record must not occur within a constant or a name, and blanks must not appear within a constant or a name.

10.9.2.1. Namelist Output Editing. Logical output constants are T for the value true and F for the value false.

Integer output constants are produced with the effect of an Iw edit descriptor.

Real constants are produced with the effect of either an F edit descriptor or an E edit descriptor, depending on the magnitude x of the value and a range $10^{d_1} \leq x < 10^{d_2}$, where d_1 and d_2 are processor-dependent integers. If the magnitude x is within this range, the constant is produced using 0PF*w.d*; otherwise, 1PE*w.d*E*e* is used.

For numeric output, reasonable processor-dependent integer values of *w*, *d*, and *e* are used for each of the cases involved.

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

Character constants produced for a file opened without a DELIM= specifier (9.3.4.9) or with a DELIM= specifier with a value of NONE:

(1) Are not delimited by apostrophes or quotation marks,

(2) Are not preceded or followed by a value separator,

(3) Have each internal apostrophe or quotation mark represented externally by one apostrophe or quotation mark, and

(4) Have a blank character inserted by the processor for carriage control at the beginning of any record that begins with the continuation of a character constant from the preceding record.

Character constants produced for a file opened with a DELIM= specifier with a value of QUOTE are delimited by quotes, are preceded and followed by a value separator, and have each internal quote represented on the external medium by two quotes.

Character constants produced for a file opened with a DELIM= specifier with a value of APOSTROPHE are delimited by apostrophes, are preceded and followed by a value separator, and have each internal apostrophe represented on the external medium by two apostrophes.

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 one or more 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 constants, each output record begins with a blank character to provide carriage control when the record is printed.

11. PROGRAM UNITS

The terms and basic concepts of program units were introduced in 2.2. A program unit may be a main program, an external subprogram, a module, or a block data program unit.

This section describes all of these program units except subprograms, which are described in Section 12.

11.1. Main Program.

R203 *main-program* **is** [*program-stmt*]
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-program-stmt

R1101 *program-stmt* **is** PROGRAM *program-name*

R1102 *end-program-stmt* **is** END [PROGRAM [*program-name*]]

Constraint: In a *main-program*, the *execution-part* must 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, must be identical to the *program-name* specified in the *program-stmt*.

The **program name** is global to the executable program, and must not be the same as the name of any other program unit, external procedure, or common block in the executable program, nor the same as any local name in the main program.

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 PROCEDURE
    . . .
END SUBROUTINE FIND
END PROGRAM ANALYSE
```

11.1.1. Main Program Specifications. The specifications in the main program must not include an OPTIONAL statement, an INTENT statement, a PUBLIC statement, a PRIVATE statement, or the equivalent attributes (5.1.2). A SAVE statement has no effect in a main program. The specification part must not declare an automatic object.

11.1.2. Main Program Executable Part. The sequence of *execution-part* statements specifies the actions of the main program during program execution. Execution of an executable program (R201) begins with the first executable construct of the main program.

A main program must not be recursive; that is, a reference to it must not appear in any program unit in the executable program, including itself.

Execution of an executable program ends with execution of the END PROGRAM statement of the main program or with execution of a STOP statement in any program unit of the executable program.

11.1.3. Main Program Internal Procedures. Any definitions of procedures internal to the main program follow the CONTAINS statement. Internal procedures are described in 11.2.1. The main program is called the **host** of its internal procedures.

11.2 Procedures. External procedures and module procedures are described in Section 12.

11.2.1 Internal Procedures. Internal procedures may appear in the main program, in an external subprogram, or in a module subprogram. Internal procedures are the same as external procedures except that the name of the internal procedure is not a global entity, an internal procedure must not contain an ENTRY statement, the internal procedure name must not be argument associated with a dummy procedure, and the internal procedure has access to host entities by host association.

11.2.2 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 as in the host and include variables, constants, procedures including interfaces, derived types, type parameters, derived-type components, range lists, and namelist groups.

Any declaration or specification of an entity with the same name as an accessible entity from the host makes the host entity inaccessible. The appearance of a name as a dummy argument or as a function result is treated as a specification.

Note that an interface block does not access the named entities from its host by host association.

11.3 Modules. A module contains specifications and definitions that are to be accessible to other program units.

R207 *module* **is** *module-stmt*
[*specification-part*]
[*module-subprogram-part*]
end-module-stmt

R1103 *module-stmt* **is** MODULE *module-name*

R1104 *end-module-stmt* **is** END [MODULE [*module-name*]]

Constraint: If the *module-name* is specified in the *end-module-stmt*, it must be identical to the *module-name* specified in the *module-stmt*.

Constraint: A module *specification-part* must not contain a *stmt-function-stmt*, an *entry-stmt*, a *format-stmt*, an *intent-stmt*, an INTENT attribute, an *optional-stmt*, or an OPTIONAL attribute.

Constraint: An automatic object must not appear in the specification part of a module (R207).

The module name is global to the executable program, and must not be the same as the name of any other program unit, external procedure, or common block in the executable program, nor the same as any local name in the module.

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 must be available. A module must 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), the 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.

In a scoping unit making reference to a module, the ONLY option on 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. The **USE statement** provides the means by which a scoping unit accesses named data objects, derived types, interface blocks, procedures, range lists, and namelist groups in a module. The entities in the scoping unit are said to be **use associated** with the entities in the module.

R1105 *use-stmt* **is** USE *module-name* [, *rename-list*]
or USE *module-name* , ONLY : [*only-list*]

R1106 *rename* **is** *use-name* = > *local-name*

R1107 *only* **is** *use-name* [= > *local-name*]

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

The USE statement without the ONLY option provides access to all public entities in the specified module.

Each *use-name* must be the name of a public entity in the module. If a *local-name* appears in a *rename-list* or an *only-list*, it is the local name for the entity specified by *use-name*; otherwise, the local name is the *use-name*.

A USE statement with the ONLY option provides access only to those entities whose names appear as *use-names* in the *only-list*. In a scoping unit, two or more accessible entities may have the same name only if no entity is referenced by this name in the scoping unit. Except for this, the local name of any entity given accessibility by a USE statement must differ from the local names of all other entities accessible to the scoping unit through USE statements and otherwise. Note that an entity may be accessed 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 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. 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.

Note that the above rule prohibits the local name from appearing in COMMON, EQUIVALENCE, and RANGE specifications, but permits the appearance of local names in NAMELIST group lists.

Examples:

```
USE STATS_LIB
```

provides access to all public entities in the module STATS__LIB.

```
USE MATH_LIB; USE STATS_LIB, PROD => SPROD
```

makes all public entities in both MATH__LIB and STATS__LIB accessible. If MATH__LIB contains an entity called PROD, it is accessible by its own name while the entity PROD of STATS__LIB is accessible by the name SPROD. Both modules may contain an entity called SUMM, for example, if SUMM does not appear in the scoping unit containing the USE statements and SUMM is not declared in a type statement in the scoping unit.

11.3.3. Examples of the Use of Modules.

11.3.3.1. 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.

11.3.3.2. Global Data. A module may contain only data objects, for example:

```
MODULE DATA_MODULE
REAL A(10), B, C(20,20)
INTEGER, DATA :: I=0
INTEGER, PARAMETER :: J=10
COMPLEX D(J,J)
END MODULE
```

Note that data objects made global in this manner may have any combination of data types.

Access to some of these may be made by a USE statement with the ONLY option, such as:

```
USE DATA_MODULE, ONLY: A, B, D
```

and access to all of them may be made by the following USE statement

```
USE DATA_MODULE
```

Access to all of them with some renaming to avoid name conflicts may be made by:

```
USE DATA_MODULE, A => AMODULE, D => DMODULE
```

11.3.3.3. Data Structures. A derived type may be defined in a module and accessed in a number of program units. This is the only way to access the same type definition in more than one program unit. For example:

```
MODULE SPARSE
TYPE NONZERO
   REAL A
   INTEGER  I, J
END TYPE
END MODULE
```

defines a type consisting of a real component and two integer components for holding the numerical value of a nonzero matrix element and its row and column indices.

11.3.3.4. 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 (INPUT, *)  N
   ALLOCATE (A (N), B (N, N), C (N, N, 2 * N))
END SUBROUTINE CONFIGURE_ARRAYS

SUBROUTINE COMPUTE
   USE WORK_ARRAYS
   !  COMPUTATIONS INVOLVING ARRAYS A, B, AND C
```

```
END SUBROUTINE COMPUTE
```

Typically, many subprograms need access to the work arrays, and all such subprograms would contain the statement

```
USE WORK_ARRAYS
```

11.3.3.5. Procedure Libraries. Interfaces to external procedures in a library may be gathered into a module. This permits the use of keyword and optional arguments, and allows static checking of the references. Different versions may be constructed for different applications, using keywords in common use in each application. An example is the following library module:

```
MODULE LIBRARY_LLS
   INTERFACE
      SUBROUTINE LLS (X, A, F, FLAG)
      REAL (*, *) X (:, :)
      REAL (*, *), ARRAY (DSIZE (X, 2)) :: A, F
      INTEGER FLAG
   END INTERFACE
END MODULE
```

This module allows the subroutine LLS to be invoked:

```
USE LIBRARY_LLS
   ...
CALL LLS (X = ABC, A = D, F = XX, FLAG = IFLAG)
   ...
```

11.3.3.6. Operator Extensions. To extend an intrinsic operator symbol to have an additional meaning, a function subprogram specifying that operator symbol in the OPERATOR option of the FUNCTION statement may be placed in a module. For example, // may be overloaded to perform concatenation of two derived-type objects serving as varying length character strings and + may be overloaded to specify matrix addition or interval arithmetic addition.

A module might contain several such functions. If the operation is written in a language other than Fortran, it may be written as an external function and its procedure interface placed in the module.

11.3.3.7. Data Abstraction. A module may encapsulate a derived-type definition and all the procedures that represent operations on values of this type. An example is given in Appendix C for set operations.

11.4. Block Data Program Units.

A block data program unit is used to provide initial values for data objects in named common blocks.

R208 *block-data* **is** *block-data-stmt*
[*specification-part*]
end-block-data-stmt

R1108 *block-data-stmt* **is** BLOCK DATA [*block-data-name*]

R1109 *end-block-data-stmt* **is** END [BLOCK DATA [*block-data-name*]]

Constraint: The *block-data-name* may be included in the *end-block-data-stmt* only if it was provided in the *block-data-stmt* and, if included, must be identical to the *block-data-name* in the *block-data-stmt*.

Constraint: A *block-data specification-part* may contain only IMPLICIT, PARAMETER, INTEGER, REAL, DOUBLE PRECISION, COMPLEX, CHARACTER, LOGICAL, COMMON, DIMENSION, EQUIVALENCE, DATA, and SAVE statements.

If an object in a named common block is initially defined, all objects having storage units in the common block storage sequence must be specified even if they are not all initially defined. More than one named common block may have objects initially defined in a single block data program unit. Note, therefore, that the primary constituents of a block data program unit are type declarations of common block objects, COMMON statements, and DATA statements.

Only an object in a named common block may be initially defined in a block data program unit. Note that objects associated with an object in a common block are considered to be in that common block.

The same named common block must not be specified in more than one block data program unit in an executable program.

There must not be more than one unnamed block data program unit in an executable program.

An example of a BLOCK DATA program unit is:

```
BLOCK DATA WORK
   COMMON /WRKCOM/ A, B, C (10, 10)
   DATA A /1.0/, B /2.0/, C /100 * 0.0/
END BLOCK DATA WORK
```

12. PROCEDURES

The concept of a procedure was introduced in 2.2.4. This section contains a complete description of procedures. The action specified by a procedure is performed when the procedure is invoked by execution of a reference to it. The reference may identify, as actual arguments, entities that are associated during execution of the procedure reference with corresponding dummy arguments in the procedure definition.

12.1. Procedure Classifications. A procedure is classified according to the form of its reference and the way it is defined.

12.1.1. Procedure Classification by Reference. The definition of a procedure specifies it to be a function or a subroutine. A reference to a function either appears explicitly as a primary within an expression, or is implied by a defined operation within an expression. A reference to a subroutine is a CALL statement or a defined assignment statement.

A procedure is classified as **elemental** if it may be referenced elementally (12.4.3, 12.4.5).

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 means other than Fortran.

An **internal procedure** is a procedure that is defined by an internal subprogram.

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.

An internal subprogram must not contain ENTRY statements.

12.1.2.3. Dummy Procedures. A dummy argument that is specified as a procedure or appears in a procedure reference is a **dummy procedure**.

12.1.2.4. Statement Functions. A function that is defined by a single statement is a **statement function**.

12.2. Characteristics of Procedures. The **characteristics of a procedure** are the classification of the procedure as a function or subroutine, the characteristics of its arguments, and the characteristics of its result value if it is a function.

12.2.1. Characteristics of Dummy Arguments. Each dummy argument is either a dummy data object, a dummy procedure, 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, type parameters (if any), shape, intent (5.1.2.3, 5.2.1), optionality (5.1.2.6, 5.2.2), and whether it is allocatable (5.1.2.4.3). 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 shape, size, or type parameters are assumed, these are characteristics.

12.2.1.2. Characteristics of Dummy Procedures. The characteristics of a dummy procedure are the explicitness of its interface (12.3.1), the characteristics of the procedure if the interface is explicit, and its optionality (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), shape, and whether it is allocatable. Where a type parameter or bound of an array is an expression, the exact dependence on other entities is a characteristic. If the length of a character data object is assumed, this is a characteristic.

12.3. Procedure Interface.

The **interface** of a procedure determines the forms of reference through which it may be invoked. The interface consists of the characteristics of the procedure, the name of the procedure, the name of each dummy argument, the defined operator (if any) by which a reference to a function may appear, and whether or not a reference to a subroutine may be implied in a defined assignment statement. The characteristics of a procedure are fixed, but the remainder of the interface may differ in different scoping units.

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 statement function is always implicit. The interface of an external procedure or dummy procedure is explicit if an interface block (12.3.2.1) for the procedure is supplied or accessible, and implicit otherwise. For example, the subroutine LLS of 11.3.3.5 has an explicit interface.

12.3.1.1. Explicit Interface. A procedure must have an explicit interface if any of the following is true:

(1) A reference to the procedure appears:

- (a) With a keyword argument (12.4.1)
- (b) As a defined assignment (subroutines only)
- (c) In an expression as a defined operator (functions only)
- (d) As an elemental reference

(2) The procedure has:

- (a) An optional dummy argument
- (b) An array-valued result (functions only)
- (c) An allocatable result (functions only)
- (d) A dummy argument that is an assumed-shape or allocatable array
- (e) A dummy argument with assumed type parameters other than character length
- (f) A result whose type parameter values are neither assumed length (character type only) nor constant.

(3) Another procedure having the same name is accessible

12.3.1.2. Implicit Interface. An actual argument may be sequence associated (12.4.1.4) with its dummy argument only if the procedure interface is implicit.

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 both.

12.3.2.1. Procedure Interface Block.

R1201 *interface-block* **is** *interface-stmt*
interface-header
[*use-stmt*]...
[*implicit-part*]
[*declaration-construct*]...
end-interface-stmt

R1202 *interface-stmt* **is** INTERFACE

R1203 *end-interface-stmt* **is** END INTERFACE

R1204 *interface-header* **is** *function-stmt*
or *subroutine-stmt*

Constraint: An *interface-block* must not contain an *entry-stmt*.

An **interface block** specifies the interface of the procedure named in the FUNCTION or SUBROUTINE statement in the interface block. The name may be that of a procedure that is defined by an entry in a subprogram.

An interface block that names as a procedure a dummy argument of the host scoping unit specifies that dummy argument to be a procedure with the specified interface. A dummy argument must not be so named more than once. Such a dummy argument may be specified in an OPTIONAL statement or with an OPTIONAL attribute in the host but must not appear in any other specification statement in the host. For example,

```
SUBROUTINE INVERSE (A, FN)
   REAL A
   INTERFACE
      FUNCTION FN (B)
      REAL FN, B
      ...
   END INTERFACE
   ...
```

specifies a subroutine whose second argument is a real function with a single real argument.

In a module, the name of a module procedure may appear in a PUBLIC or PRIVATE statement or be given the equivalent attribute, but must not appear in any other specification statement.

The characteristics (12.2) of the procedure itself must be identical with those specified by the interface block. The presence of the interface block does not require the availability of the procedure until it is invoked. Within a scoping unit, only one interface block may be provided for a particular procedure. If the procedure is a module procedure, an external procedure, or an internal procedure, the names of the arguments and the operator (if present) override those of the procedure itself.

12.3.2.2. EXTERNAL Statement. An **EXTERNAL statement** is used to specify a name as representing an external procedure or dummy procedure, and to permit such a name to be used as an actual argument.

R1205 *external-stmt* **is** EXTERNAL *external-name-list*

Constraint: Each *external-name* must 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. Appearance of an intrinsic procedure name in an EXTERNAL statement causes that name to become the name of some external subprogram and an intrinsic procedure of the same name is not available in the scoping unit.

Only one appearance of a name in all of the EXTERNAL statements in any one sequence of specification part statements is permitted.

An example of an EXTERNAL statement is:

```
SUBROUTINE SUB (FOCUS)
EXTERNAL FOCUS
```

12.3.2.3. INTRINSIC Statement. An **INTRINSIC statement** is used to specify a name as representing an intrinsic procedure (Section 13). It also permits a name that represents a specific intrinsic function to be used as an actual argument.

R1206 *intrinsic-stmt* **is** INTRINSIC *intrinsic-procedure-name-list*

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 function name (13.1) in an INTRINSIC statement does not cause that name to lose its generic property.

Only one appearance of a name in all of the INTRINSIC statements in any one sequence of specification part statements is permitted. Note that a name must not appear in both an EXTERNAL and an INTRINSIC statement in the same sequence of specification part statements.

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 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 a procedure is implicit must 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:

R1207 *function-reference* **is** *function-name* ([*actual-arg-spec-list*])

Constraint: The *actual-arg-spec-list* for a function reference must not contain an *alt-return-spec*.

R1208 *call-stmt* **is** CALL *subroutine-name* [([*actual-arg-spec-list*])]

12.4.1. Actual Argument List.

R1209 *actual-arg-spec* **is** [*keyword* =] *actual-arg*

R1210 *keyword* **is** *dummy-arg-name*

R1211 *actual-arg* **is** *expr*
or *variable*
or *procedure-name*
or alt-return-spec

R1212 *alt-return-spec* is * label

Constraint: The *keyword* may be omitted from an *actual-arg-spec* only if the *keyword* has been omitted from each preceding *actual-arg-spec* in the argument list.

Constraint: Each *keyword* must be the name of a dummy argument in the interface of the procedure.

Constraint: A *procedure-name actual-arg* must not be the name of an internal procedure and must not be the name of an intrinsic subroutine (13.8). If it is the name of an intrinsic function, it must be the specific name for the function (13.1).

Constraint: The *label* used in the *alt-return-spec* must be the statement label of a branch target statement that appears in the same scoping unit as the *call-stmt*.

In either a subroutine reference or a function reference, the actual argument list identifies the correspondence between the actual arguments supplied and the dummy arguments of the procedure. In the absence of a keyword, an actual argument is associated with the dummy argument occupying the corresponding position in the dummy argument list; i.e., the first actual argument is associated with the first dummy argument, the second actual argument is associated with the second dummy argument, etc. If a keyword is present, the actual argument is associated with the dummy argument whose name is the same as the keyword. Exactly one actual argument must be associated with each nonoptional dummy argument. At most one actual argument may be associated with each optional argument. Each actual argument must be associated with a dummy argument. For example, the procedure

```
SUBROUTINE SOLVE (FUNCT, SOLUTION, METHOD, STRATEGY, PRINT)
   INTERFACE
      FUNCTION FUNCT (X)
      REAL FUNCT, X
   END INTERFACE
   REAL SOLUTION
   INTEGER, OPTIONAL :: METHOD, STRATEGY, PRINT
   ...
```

may be invoked with

```
CALL SOLVE (FUN, SOL, PRINT = 6)
```

12.4.1.1. Arguments Associated with Dummy Data Objects. If a dummy argument is a dummy data object, the associated actual argument must be an expression of the same type or a data object of the same type. The type parameter values of the actual argument, if any, must agree with or be assumed by the dummy argument, except that a character actual argument may have a length greater than that of the character dummy argument if the interface is implicit, in which case, the effect is as if the leftmost substring of appropriate length were used as the actual argument. The shape of the actual argument must agree with or be assumed by the dummy argument except when a procedure reference is elemental (12.4.3, 12.4.5) or when the actual argument is sequence associated with the dummy argument (12.4.1.4). 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 (6.2.4.2). Changing the effective range of a dummy argument array has no effect on the effective range of the associated actual argument array.

If the dummy argument is allocatable, the actual argument must be an allocatable array that does not have the RANGE attribute and the types, type parameter values, if any, and ranks must agree. The allocation status of the dummy argument becomes that of the actual argument at invocation of the procedure. This may be changed during execution of the procedure, in which case the actual argument allocation state becomes that of the dummy argument when the procedure completes execution.

If the intent of a dummy argument is OUT or INOUT, the actual argument must be definable. If the intent of a dummy argument is OUT, the corresponding actual argument becomes undefined at the time the association is established.

12.4.1.2 Arguments Associated with Dummy Procedures. If a dummy argument is a dummy procedure, the associated actual argument must be the name of an external, module, dummy, or intrinsic procedure. The only intrinsic procedures permitted are those listed in 13.11 and not marked with a bullet (•). The actual argument name must be one for which exactly one procedure is accessible in the invoking scoping unit. (A specific intrinsic function and a generic intrinsic function of the same name are considered to be one procedure.) The actual argument procedure must not have dummy arguments with assumed type parameters other than character assumed lengths.

If the interface of the dummy procedure is explicit, the characteristics of the associated procedure must be the same as the characteristics of the dummy procedure (12.2).

If the interface of the dummy procedure is implicit and either the name of the dummy procedure is explicitly typed or the procedure is referenced as a function, the dummy procedure must not be referenced as a subroutine and the actual argument must be a function or dummy procedure.

If the interface of the dummy procedure is implicit and a reference to the procedure appears as a subroutine reference, the actual argument must be a subroutine or dummy procedure.

12.4.1.3 Arguments Associated with Alternate Return Indicators. If a dummy argument is an asterisk (12.5.2.3), the associated actual argument must be an alternate return specifier. The label in the alternate return specifier must identify an executable construct in the scoping unit containing the procedure reference.

12.4.1.4 Sequence Association. A **sequence array** is an array without the RANGE attribute that is an allocatable array, assumed-size array, or explicit-shape array that is either a dummy array associated with a sequence array or is not a dummy argument. An actual argument represents an **element sequence** if it is a whole array name, array element designator, or array element substring designator and the array is a sequence array. If the actual argument is a whole array name, 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 an array element substring designator, the element sequence consists of the character storage units beginning with the first storage unit in that array element substring and continuing to the end of the array. The character storage units are viewed as elements consisting of consecutive groups of character storage units having the length of the array element substring. Thus, the first such element is the array element substring itself. Note that some of the elements in the element sequence may consist of storage units from different elements of the original array.

If the interface for a procedure reference is implicit, the actual argument represents an element sequence, and the corresponding dummy argument is an array-valued data object that is neither allocatable nor assumed shape, the actual argument is **sequence associated** with the dummy argument. The rank and shape of the actual argument need not agree with the rank and shape of the dummy argument, but the number of elements in the dummy argument must not exceed the number of elements in the element sequence of the actual argument. If the dummy argument is assumed size, the number of elements in the dummy argument is exactly the number of elements in the element sequence.

The effects of the shape matching rules in 12.4.1.1 and 12.4.1.4 for nonelemental references and of the explicit interface requirements in 12.3.1.1 are in Table 12.1.

Table 12.1. Shape Matching Rules for Nonelemental References.

Dummy Argument	*Actual Argument*: Nonsequence Array (Including ranged or allocatable)	Allocatable and Not Ranged	Sequence Array, Not Assumed-Size, Not Allocatable	Assumed-Size	Element of Sequence Array	Other Scalars (Including element of nonsequence array)
Explicit-Shaped	Allowed	Allowed, Shape may differ	Allowed, Shape may differ	Allowed, Shape may differ	Allowed, Shape may differ	Not allowed
Assumed-Size	Not allowed	Allowed, Shape may differ	Allowed, Shape may differ	Allowed, Shape may differ	Allowed, Shape may differ	Not allowed
Assumed-Shape	Allowed, Explicit interface required	Allowed, Explicit interface required	Allowed, Explicit interface required	Not allowed	Not allowed	Not allowed
Allocatable	Not allowed	Allowed, Explicit interface required	Not allowed	Not allowed	Not allowed	Not allowed
Scalar	Not allowed	Not allowed	Not allowed	Not allowed	Allowed	Allowed

Notes for Table 12.1:

(1) For arrays of type character, "element" includes element substrings.

(2) "Shape may differ" indicates that the shape of the actual argument need not match the shape of the dummy argument if the interface is implicit.

12.4.2. Function Reference. A function is invoked during expression evaluation by a function reference or by defined operations (7.1.3). When it is invoked, all actual argument expressions are evaluated, then the arguments are associated, and then the function is executed. When execution of the function is complete, the value of the function result is available for use in the expression that caused the function to be invoked. The type, type parameters (if any), and the shape of the result are determined by the interface of the function, specifically the characteristics of the function result (12.2.2).

12.4.3. Elemental Function Reference. A reference to a function is an **elemental reference** if the interface for the function is explicit, if its dummy arguments and result are all scalar data objects, if the type parameters of the result are independent of the values of the actual arguments, and if the function is not a dummy procedure. Arguments to such a reference may be arrays, provided all array arguments have the same effective shape. The result has the same shape as the array arguments and the value of each element in the result is obtained by evaluating the function using the scalar arguments and the corresponding elements of the array arguments. For example, if X and Y are arrays of shape [*m, n*],

```
MAX (X, 0.0, Y)
```

is an array expression of shape [*m, n*] whose elements have values

52 MAX(X(*i*, *j*), 0.0, Y(*i*, *j*)), *i* = 1, 2, ..., *m*, *j* = 1, 2, ..., *n*

The result must not depend on the order in which these references are made.

For example, the reference to the procedure

```
FUNCTION SCALE (A)
   READ (*, *) FACTOR
   SCALE = A * FACTOR
END
```

must not be an elemental reference, because the elements of the array result depend on the order in which the scalars FACTOR are read.

A function reference is not interpreted as being an elemental reference if it may be interpreted as a nonelemental reference to a function with the same name whose interface is explicit in the scoping unit containing the reference. For example, the expression POWER (A (1 : 10)), where A is an array, would not be interpreted as an elemental reference if a function POWER with one argument having the type and rank of A is accessible.

12.4.4. Subroutine Reference. A subroutine is invoked by execution of a CALL statement or defined assignment statement (7.5.1.3). When a subroutine is invoked, all actual argument expressions are evaluated, then the arguments are associated, and then the subroutine is executed. When the action specified by the subroutine is completed, execution of the CALL statement or defined assignment statement is also completed. If a CALL statement includes one or more alternate return specifiers among its arguments, control may be transferred to one of the statements indicated depending on the action specified by the subroutine.

12.4.5. Elemental Assignment. A reference to an assignment subroutine may be an elemental reference in a defined assignment statement if its dummy arguments are scalar and the type parameters of the first dummy argument are independent of the value of the second dummy argument. In such a reference, the first actual argument is array valued and the second is of the same effective shape or is scalar. The subroutine is invoked once for each element of the first actual argument, using the corresponding element of the second actual argument or its scalar value. The result must not depend on the order in which these invocations are made. An assignment is not interpreted as an elemental assignment if it may be interpreted as a nonelemental assignment.

12.5. Procedure Definition.

12.5.1. Intrinsic Procedure Definition. Intrinsic procedures are defined as an inherent part of the processor. A standard-conforming processor must include the intrinsic procedures described in Section 13, but may include others. However, a standard-conforming program must not make use of intrinsic procedures other than those described in Section 13.

12.5.2. Procedures Defined by Subprograms. When a procedure defined by a subprogram is invoked, an instance (12.5.2.4) of the procedure is created and executed. Execution begins with the first executable construct following the FUNCTION, SUBROUTINE, or ENTRY statement specifying the name of the procedure invoked.

12.5.2.1. Effects of Intent on Subprograms. The intent of dummy data objects limits the way in which they may be used in a subprogram. A dummy data object having intent IN may not be defined or redefined by the subprogram. A dummy data object having intent OUT is initially undefined in the subprogram. A dummy data object with intent INOUT may be referenced or be defined. A dummy data object whose intent is neither specified nor implied by the presence of the OPERATOR or ASSIGNMENT option is subject to the limitations of the data entity that is the associated actual argument. That is, a reference to the dummy data object may appear if the actual argument is defined and may be defined if the actual

argument is definable.

12.5.2.2. Function Subprogram.

R204 *external-subprogram* **is** *procedure-heading*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
procedure-ending

R205 *procedure-heading* **is** *function-stmt*
or *subroutine-stmt*

R206 *procedure-ending* **is** *end-function-stmt*
or *end-subroutine-stmt*

R1213 *function-stmt* **is** [*prefix*] FUNCTION *function-name* ■
■ ([*dummy-arg-name-list*]) [*suffix*]

R1214 *prefix* **is** *type-spec* [RECURSIVE]
or RECURSIVE [*type-spec*]

R1215 *suffix* **is** RESULT (*result-name*) [OPERATOR (*defined-operator*)]
or OPERATOR (*defined-operator*) [RESULT (*result-name*)]

R1216 *end-function-stmt* **is** END [FUNCTION [*function-name*]]

Constraint: FUNCTION must be present on the *end-function-stmt* of an internal or module function.

Constraint: An internal function must not contain an ENTRY statement.

Constraint: If *function-name* is supplied on the *end-function-stmt*, it must agree with the *function-name* on 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 function name appearing in a type statement in the declaration part of the function subprogram. It must not be specified both ways. If it is not specified either way, it is determined by the implicit typing rules in force within the function subprogram. If the function result is array-valued or allocatable, this must be specified by specifications of the function name within the function body. The specifications of the function result attribute, the specification of dummy argument attributes, and the information in the procedure heading collectively define the interface of the function (12.3).

The keyword RECURSIVE must be present if the function invokes itself, either directly or indirectly.

The name of the function is *function-name*.

If RESULT is specified, the name of the result variable of the function is *result-name* 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 *result-name* must not appear in any specification statement.

If OPERATOR is specified, the interface for the procedure includes the ability to invoke it using a defined operator. This operator must be unary if the function has one dummy argument and binary if it has two dummy arguments; no other number of dummy arguments is permitted. The dummy arguments must be nonoptional dummy data objects with intent IN. If the intent of a dummy argument is not specified, the specification of OPERATOR causes it to have intent IN. If the operator is unary, in any scoping unit in which this interface is explicit, any reference to that operator in which the operand has the characteristics corresponding to the dummy argument of the function is treated as a reference to the function. If the operator

is binary, in any scoping unit in which this interface is explicit, any reference to that operator in which the left operand has the characteristics corresponding to the first dummy argument of the function and the right operand has the characteristics corresponding to the second dummy argument of the function is treated as a reference to the function. Whether unary or binary, any such reference may be elemental (12.4.3).

An example of a recursive function is:

```
RECURSIVE INTEGER FUNCTION BEST (A, B) RESULT (BST)
   INTEGER A, B
   ...
   BST = BEST (A - 1, B - 1)
END FUNCTION BEST
```

12.5.2.3. Subroutine Subprogram.

R204 *external-subprogram* **is** *procedure-heading*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
procedure-ending

R205 *procedure-heading* **is** *function-stmt*
or *subroutine-stmt*

R206 *procedure-ending* **is** *end-function-stmt*
or *end-subroutine-stmt*

R1217 *subroutine-stmt* **is** [RECURSIVE] SUBROUTINE *subroutine-name* ■
■ [([*dummy-arg-list*])] [ASSIGNMENT]

R1218 *dummy-arg* **is** *dummy-arg-name*
or *

R1219 *end-subroutine-stmt* **is** END [SUBROUTINE [*subroutine-name*]]

Constraint: SUBROUTINE must be present on the END statement of an internal or module subroutine.

Constraint: An internal subroutine must not contain an ENTRY statement.

Constraint: If *subroutine-name* is present on the *end-subroutine-stmt*, it must agree with the *subroutine-name* on the *subroutine-stmt*.

The keyword RECURSIVE must be present if the subroutine subprogram invokes itself, either directly or indirectly.

If ASSIGNMENT is specified, the subroutine may be referenced as an assignment statement and is called an **assignment subroutine**. The subroutine must have exactly two arguments which must be nonoptional dummy data objects. The first dummy argument must have intent OUT or INOUT. If its intent is not specified, it has intent OUT. The second dummy argument must have intent IN. If its intent is not specified, it has intent IN. In any program unit in which this interface is explicit, any assignment statement in which the variable has the characteristics corresponding to the first dummy argument of the subroutine and the expression has the characteristics corresponding to the second dummy argument of the subroutine is treated as a reference to the subroutine. Note also the possibility of an assignment statement referencing the subroutine elementally (12.4.5).

An example of an assignment subroutine is:

```
SUBROUTINE ASSIGNSC (STR, CHAR) ASSIGNMENT
! A MODULE PROCEDURE FOR ASSIGNING A CHARACTER VARIABLE
! TO A STRING VARIABLE OF THE TYPE DEFINED IN 4.4.1.1,
! ASSUMING THAT THE TYPE DEFINITION IS ALSO IN THE MODULE.
```

```
TYPE (STRING (*)) STR
CHARACTER (LEN = *) CHAR
   STR % LENGTH = LEN (CHAR)
   STR % VALUE = CHAR
END SUBROUTINE ASSIGNSC
```

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.

Each instance has an independent sequence of execution and an independent set of dummy arguments and nonsaved data objects. If an internal procedure or statement function contained in the subprogram is invoked directly from an instance of the subprogram, the created instance of that internal procedure or statement function also has access to the entities of that instance of the host subprogram.

All other entities, including saved data objects, are common to all instances of the subprogram. For example, the value of a saved data object appearing in one instance may have been defined in a previous instance or by a DATA attribute or statement.

12.5.2.5 ENTRY Statement.

R1220 *entry-stmt* **is** ENTRY *entry-name* [([*dummy-arg-list*])]

Constraint: A *dummy-arg* may be an alternate return indicator only if the ENTRY statement is contained in a subroutine subprogram.

If the ENTRY statement is contained in a function subprogram, an additional function is defined by that subprogram. The name of the function and its result variable is *entry-name*. The characteristics of the function result are specified by specifications of *entry-name*. The dummy arguments of the function are those specified on the ENTRY statement. If the characteristics of the result of the function named on the ENTRY statement are the same as the characteristics of the function named on the FUNCTION statement, their result variables are associated. Otherwise, they are storage associated with the restrictions that they are scalar, that they have type and type parameters permitting storage association, and that they have the same lengths if they are of character type.

If the ENTRY statement is contained in a subroutine subprogram, an additional subroutine is defined by that subprogram. The name of the subroutine is *entry-name*. The dummy arguments of the subroutine are those specified on the ENTRY statement.

12.5.2.6 RETURN Statement.

R1221 *return-stmt* **is** RETURN [*scalar-int-expr*]

Constraint: The *return-stmt* must be contained 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 nth alternate return specifier in the actual argument list. If the expression is omitted or has a value outside the required range, there is no transfer of control to an alternate return.

Execution of an END statement, END FUNCTION statement, or END SUBROUTINE statement is equivalent to executing a RETURN statement with no expression.

12.5.2.7 CONTAINS Statement.

R1222 *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.

12.5.2.8 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 procedure or is not a dummy argument of the invoking procedure. A dummy argument that is not optional must 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 must not be referenced or be defined.

(2) If it is a dummy procedure, it must not be invoked.

(3) It must not be supplied as an actual argument corresponding to a nonoptional dummy argument other than the argument of the PRESENT intrinsic function.

(4) It may be supplied as an actual argument corresponding to an optional dummy argument. The optional dummy argument is then also considered not to be associated with an actual argument.

12.5.2.9 Restrictions on Entities Associated with Dummy Arguments. While an entity is associated with a dummy argument, the following restrictions hold:

(1) No action may be taken that affects the value or availability of the entity or any part of it, except through the dummy argument. For example, in

```
SUBROUTINE OUTER
REAL, ALLOCATABLE :: A (:)
...
ALLOCATE (A (1:N))
...
CALL INNER (A)
...
CONTAINS
   SUBROUTINE INNER (B)
   REAL :: B (:)
   ...
   END SUBROUTINE INNER

   SUBROUTINE SET (C, D)
      REAL, OUT :: C
      REAL, IN :: D
      C = D
      END SUBROUTINE SET
END SUBROUTINE OUTER
```

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 availability of A without using B. In this case,

```
DEALLOCATE (B)
```

also would not be permitted, but would be permitted if B were declared ALLOCATABLE.

Note that if there is a partial or complete overlap between the actual arguments associated with two different dummy arguments of the same procedure, the overlapped portions are unchangeable during the execution of the procedure. For example, in

```
CALL SUB (A (1:5), A (3:9))
```

A (3:5) cannot be changed through the first argument because it is part of the argument associated with the second dummy argument and cannot be changed through the second dummy argument because it is part of the argument associated with the first dummy argument. A (1:2) remains changeable through the first dummy argument and A (6:9) remains changeable through the the second dummy argument.

Note that since a dummy argument declared with an intent of IN cannot be used to change the associated actual argument, the associated actual argument remains constant throughout the execution of the procedure.

(2) If any part of the entity is defined 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. For example, 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 must 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.

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 must not be contained in a Fortran program unit and a Fortran program unit must not be contained in the definition of a non-Fortran procedure. The interface to a non-Fortran procedure may be specified in an interface block.

12.5.4 Statement Function. A statement function is a function defined by a single statement.

R1223 *stmt-function-stmt* **is** *function-name* ([*dummy-arg-name-list*]) = *expr*

Constraint: The *expr* may be composed only of constants (literal and named), references to scalar variables and array elements, references to functions and function dummy procedures, and intrinsic operators. If a reference to another statement function appears in *expr*, its definition must have been provided earlier in the scoping unit.

Constraint: The *function-name* and each *dummy-arg-name* must be specified, explicitly or implicitly, to be scalar data objects.

The dummy arguments have a scope of the statement function statement. A given name may appear only once in any statement function dummy argument list.

Each scalar variable reference may be either a reference to a dummy argument of the statement function or a reference to a variable within the same scoping unit as the statement function statement.

The statement function produces the same result value as an internal function of the form:

FUNCTION *function-name* ([*dummy-arg-name-list*])
function-and-dummy-specifications
function-name = *expr*
END FUNCTION *function-name*

where *function-and-dummy-specifications* are the specifications necessary to cause *function-name* and each *dummy-arg-name* to be given explicitly the same type and type parameters that those names are given explicitly in the scoping unit containing the statement function. Note, however, that unlike the internal function, the statement function always has an implicit interface and may not be supplied as a procedure argument.

12.5.5 Overloading Names. Two or more functions may be accessible with the same name in the same scoping unit if any argument list would be appropriate in referencing at most one of them. Similarly, two or more functions may be accessible with the same operator symbol in the same scoping unit, two or more subroutines may be accessible with the same name in the same scoping unit, and two or more subroutines may be accessible as assignments in the same program scope (Section 14). The specific rules on how any two such procedures must differ are given in 14.1.2.3.

13 INTRINSIC PROCEDURES

13.1 Intrinsic Functions. An **intrinsic function** is an inquiry function, an elemental function, or a transformational function. An **inquiry function** is one whose result depends on the explicit or implicit declarations associated with its principal argument and not on the value of this argument; in fact, the argument value may be undefined. An **elemental function** is one that is specified for scalar arguments, but may be applied to array arguments as described in 13.2. All other intrinsic functions are **transformational functions**; they almost all have one or more array-valued arguments or an array-valued result.

Generic names of intrinsic functions are listed in 13.9.1 through 13.9.14. In most cases, generic functions accept arguments of more than one type and the type of the result is the same as the type of the arguments. **Specific names** of intrinsic functions with corresponding generic names are listed in 13.11.

If an intrinsic function is used as an actual argument to an external procedure, its specific name must be used and it may be referenced in the external procedure only with scalar arguments. If an intrinsic function does not have a specific name, it must not be used as an actual argument.

13.2 Elemental Intrinsic Function Arguments and Results. If a generic name or a specific name is used to reference an elemental intrinsic function, the shape of the result is the same as the effective shape of the argument with the greatest rank. If the arguments are all scalar, the result is scalar. For those elemental intrinsic functions that have more than one argument, all arguments must be conformable. In the array-valued case, the values of the elements of the result are the same as would have been obtained if the scalar-valued function had been applied separately to corresponding elements of each argument.

All intrinsic procedures may be invoked with either positional or keyword arguments. The descriptions in 13.12 give the keyword names and positional sequence. A keyword is required for an argument only if a preceding optional argument is omitted.

13.3 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.

13.4 Numeric, Mathematical, Character, and Derived-Type Functions.

13.4.1 Numeric Functions. The elemental functions INT, REAL, DBLE, and CMPLX perform type conversions. The elemental functions AIMAG, CONJG, AINT, ANINT, NINT, ABS, MOD, SIGN, DIM, DPROD, MAX, and MIN perform simple numeric operations.

13.4.2 Mathematical Functions. The elemental functions SQRT, EXP, LOG, LOG10, SIN, COS, TAN, ASIN, ACOS, ATAN, ATAN2, SINH, COSH, and TANH evaluate elementary mathematical functions.

13.4.3 Character Functions. The elemental functions ICHAR, CHAR, LGE, LGT, LLE, LLT, IACHAR, ACHAR, INDEX, VERIFY, ADJUSTL, ADJUSTR, REPEAT, SCAN, and LEN_TRIM perform character operations. The TRIM function returns the argument with trailing blanks removed.

13.4.4 Character Inquiry Function. The inquiry function LEN returns the length of a character entity. The value of the argument to this function 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.

13.4.5 Derived-Type Inquiry Functions. A derived-type definition that includes a dummy type parameter list causes the implicit definition of a set of inquiry functions, one for each type parameter. For nonprecision parameters, these inquiry functions have names that are the same as the dummy parameter names. For precision and exponent range parameters. the inquiry functions are called EFFECTIVE_PRECISION and EFFECTIVE_EXPONENT_RANGE. Each has a single argument whose type must be that defined by the type definition and returns a single integer result. For the inquiry functions of nonprecision parameters, the result is the value of the indicated parameter for the structure that is the argument.

The result of the EFFECTIVE_PRECISION and EFFECTIVE_EXPONENT_RANGE functions when applied to a structure of a type with precision and exponent range parameters is the same as would be obtained if these functions were applied to an object of type real declared with the same precision and exponent range parameter values. For example, consider a type defined by:

```
TYPE MATRIX (PRECISION, EXPONENT_RANGE, ORDER)
   REAL (PRECISION, EXPONENT_RANGE), ARRAY (ORDER, ORDER) :: A
END TYPE MATRIX
```

The implicitly-defined inquiry functions that take type MATRIX arguments are ORDER, EFFECTIVE_PRECISION, and EFFECTIVE_EXPONENT_RANGE. If the following objects were declared:

```
TYPE (MATRIX (10, 50, 25)) :: COVAR
TYPE (MATRIX (5, 30, 3)) :: ROTATE
REAL (10, 50) :: X
REAL (5, 30) :: Y
```

references to these inquiry functions would return the following results:

ORDER (COVAR) returns 25
ORDER (ROTATE) returns 3
EFFECTIVE_PRECISION (COVAR)
    returns the same value as EFFECTIVE_PRECISION (X)
EFFECTIVE_EXPONENT_RANGE (COVAR)
    returns the same value as EFFECTIVE_EXPONENT_RANGE (X)
EFFECTIVE_PRECISION (ROTATE)
    returns the same value as EFFECTIVE_PRECISION (Y)
EFFECTIVE_EXPONENT_RANGE (ROTATE)
    returns the same value as EFFECTIVE_EXPONENT_RANGE (Y)

The scope of these implicitly defined inquiry functions is the same as that of the derived type. These functions may be referenced in any scoping unit in which the derived-type definition may be referenced. Note that the argument need not be defined at the time the function is referenced. For example, if

```
TYPE (STRING (100)) :: LINE
```

declares an object of the type STRING as defined in 4.4.1.1, the function reference MAX_SIZE (LINE) returns the integer result 100.

13.5 Transfer Function. The function TRANSFER serves to gain access to the physical representation specified by its first argument in a form specified by its second argument.

13.6 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 executable program is executed.

13.6.1 **Models for Integer and Real Data.** The model set for integer i is defined by:

$$i = s \times \sum_{k=1}^{q} w_k \times r^{k-1}$$

where r is an integer exceeding one, q is a positive integer, each w_k is a nonnegative integer less than r, and s is $+1$ or -1. The model set for real x is defined by:

$$x = \begin{cases} 0 \text{ or} \\ s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k}, \end{cases}$$

where b and p are integers exceeding one; each f_k is a nonnegative integer less than b, except f_1 which is also nonzero; s is $+1$ or -1; and e is an integer that lies between some integer maximum e_{max} and some integer minimum e_{min} inclusively. 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 integer and each real data type implemented by the processor. The parameters characterize the set of available numbers in the definition of the model. The floating point manipulation and inquiry functions provide values related to the parameters and other constants related to them. Examples of these functions in this section use the models:

$$i = s \times \sum_{k=1}^{31} w_k \times 2^{k-1}$$

and

$$x = 0 \text{ or } s \times 2^e \times \left(1/2 + \sum_{k=2}^{24} f_k \times 2^{-k} \right), \quad -126 \le e \le 127$$

13.6.2 Numeric Inquiry Functions. The inquiry functions RADIX, DIGITS, MINEXPONENT, MAXEXPONENT, EFFECTIVE_PRECISION, EFFECTIVE_EXPONENT_RANGE, HUGE, TINY, and EPSILON return scalar values related to the parameters of the model associated with the type and type parameters of the arguments. The value of the arguments to these functions need not be defined, the shape of array arguments need not be defined, and array arguments need not be allocated.

It is not necessary for a processor to evaluate the arguments of a numeric inquiry function if the value of the function can be determined otherwise.

13.6.3 Floating Point Manipulation Functions. The elemental functions EXPONENT, SCALE, NEAREST, FRACTION, SETEXPONENT, SPACING, and RRSPACING return values related to the components of the model values (13.6.1) associated with the actual values of the arguments.

13.7 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.

13.7.1 The Shape of Array Arguments. The transformational array intrinsic functions operate on each array argument as a whole. The effective shape of the corresponding actual argument must therefore be defined; that is, the actual argument must be an array section, an assumed-shape array, an explicit-shape array, an allocatable array that has been allocated,

an alias array that is alias associated, or an array-valued expression. It must not be an assumed-size array.

Some of the inquiry intrinsic functions accept array arguments for which the shape need not be defined. Assumed-size arrays may be used as arguments to these functions; they include the functions ELBOUND and DLBOUND, and certain references to DSIZE, ESIZE, EUBOUND, and DUBOUND.

13.7.2 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 must be of type LOGICAL.

13.7.3 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 DOTPRODUCT operates on two vectors and returns their scalar product. The vectors are of the same type (numeric or logical) as for MATMUL. For logical vectors, DOTPRODUCT returns the Boolean scalar product.

13.7.4 Array Reduction Functions. The array reduction functions SUM, PRODUCT, MAXVAL, MINVAL, COUNT, ANY, and ALL perform numerical, logical, and counting operations on arrays. They may be applied to the whole array to give a scalar result or they may be applied over a given dimension to yield a result of rank reduced by one. By use of a logical mask that is conformable with the given array, the computation may be confined to any subset of the array (e.g., the positive elements).

13.7.5 Array Inquiry Functions. The functions ESIZE, ESHAPE, ELBOUND, and EUBOUND return, respectively, the effective number of elements, the effective shape, and the effective lower and upper bounds of the subscripts along each dimension. The functions DSIZE, DSHAPE, DLBOUND, and DUBOUND return, respectively, the declared size of the array, the declared shape, and the declared lower and upper bounds of the subscripts along each dimension.

The values of the array arguments to these functions need not be defined.

It is not necessary for a processor to evaluate the arguments of an array inquiry function if the value of the function can be determined otherwise.

13.7.6 Array Construction Functions. The functions MERGE, SPREAD, RESHAPE, PACK, and UNPACK construct new arrays from the elements of existing ones. MERGE combines two conformable arrays into one by an element-wise choice based on a logical mask. SPREAD 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). RESHAPE produces an array with the same elements and a different shape. 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.

13.7.7 Array Manipulation Functions. The functions TRANSPOSE, EOSHIFT, and CSHIFT manipulate arrays. TRANSPOSE performs the matrix transpose operation on a two-dimensional array. The shift functions leave the shape of an array unaltered but shift the positions of the elements parallel to a specified dimension of the array. These shifts are either circular (CSHIFT), in which case elements shifted off one end reappear at the other end, or end-off (EOSHIFT), in which case specified boundary elements are shifted into the vacated positions.

The functions MAXLOC and MINLOC return the location (subscripts) of an element of an array that has maximum and minimum values, respectively. By use of an optional logical mask that is conformable with the given array, the reduction may be confined to any subset of the array.

13.8 Intrinsic Subroutines.

Intrinsic subroutines are supplied by the processor and have the special definitions given in 13.10 and 13.12. An intrinsic subroutine is referenced by a CALL statement that uses its name explicitly. The name of an intrinsic subroutine must not be used as an actual argument. The effect of a subroutine reference is as specified in 13.12.

13.8.1 Date and Time Subroutines. The subroutines DATE_AND_TIME and SYSTEM_CLOCK return integer data from the date and real-time clock. The time returned is local, but there are facilities for finding out the difference between local time and Greenwich Mean Time.

13.8.2 Pseudorandom Numbers. The subroutine RANDOM returns a pseudorandom number or an array of pseudorandom numbers. The subroutine RANDOMSEED initializes or restarts the pseudorandom number sequence.

13.9 Tables of Generic Intrinsic Functions.

In both the following table and in the individual descriptions of the intrinsic procedures, the arguments shown are the names that must be used when passing actual arguments using the keyword form. For example, a reference to CMPLX may be written in the form CMPLX (TARGET, SOURCE, M) or in the form CMPLX (Y = SOURCE, MOLD = M, X = TARGET).

Many of the keyword arguments have names that are indicative of their usage. For example:

MOLD	Describes the characteristics of an argument or 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

13.9.1 Argument Presence Inquiry Function.

PRESENT (A)	Argument presence

13.9.2 Numeric Functions.

ABS (A)	Absolute value
AIMAG (Z)	Imaginary part of a complex number
AINT (A)	Truncation to whole number
ANINT (A)	Nearest whole number
CMPLX (X, Y, MOLD) Optional Y, MOLD	Conversion to complex type
CONJG (Z)	Conjugate of a complex number
DBLE (A)	Conversion to double precision real type
DIM (X, Y)	Positive difference
DPROD (X, Y)	Double precision real product

1	INT (A)	Conversion to integer type
2	MAX (A1, A2, A3,...)	Maximum value
3	Optional A3,...	
4	MIN (A1, A2, A3,...)	Minimum value
5	Optional A3,...	
6	MOD (A, P)	Remainder modulo P
7	NINT (A)	Nearest integer
8	REAL (A, MOLD)	Conversion to real type
9	Optional MOLD	
10	SIGN (A, B)	Transfer of sign

11 **13.9.3 Mathematical Functions.**

12	ACOS (X)	Arccosine
13	ASIN (X)	Arcsine
14	ATAN (X)	Arctangent
15	ATAN2 (Y, X)	Arctangent
16	COS (X)	Cosine
17	COSH (X)	Hyperbolic cosine
18	EXP (X)	Exponential
19	LOG (X)	Natural logarithm
20	LOG10 (X)	Common logarithm (base 10)
21	SIN (X)	Sine
22	SINH (X)	Hyperbolic sine
23	SQRT (X)	Square root
24	TAN (X)	Tangent
25	TANH (X)	Hyperbolic tangent

26 **13.9.4 Character Functions.**

27	ACHAR (I)	Character in given position
28		in ASCII collating sequence
29	ADJUSTL (STRING)	Adjust left
30	ADJUSTR (STRING)	Adjust right
31	CHAR (I)	Character in given position
32		in processor collating sequence
33	IACHAR (C)	Position of a character
34		in ASCII collating sequence
35	ICHAR (C)	Position of a character
36		in processor collating sequence
37	INDEX (STRING, SUBSTRING, BACK)	Starting position of a substring
38	Optional BACK	
39	LEN_TRIM (STRING)	Length without trailing blank characters
40	LGE (STRING_A, STRING_B)	Lexically greater than or equal
41	LGT (STRING_A, STRING_B)	Lexically greater than
42	LLE (STRING_A, STRING_B)	Lexically less than or equal
43	LLT (STRING_A, STRING_B)	Lexically less than
44	REPEAT (STRING, NCOPIES)	Repeated concatenation
45	SCAN (STRING, SET, BACK)	Scan a string for a character in a set
46	Optional BACK	
47	TRIM (STRING)	Remove trailing blank characters
48	VERIFY (STRING, SET, BACK)	Verify the set of characters in a string
49	Optional BACK	

50 **13.9.5 Character Inquiry Functions.**

51	LEN (STRING)	Length of a character entity

13.9.6 Numeric Inquiry Functions.

DIGITS (X)	Number of significant digits in the model
EFFECTIVE__EXPONENT__RANGE (X)	Effective decimal exponent range
EFFECTIVE__PRECISION (X)	Effective decimal precision
EPSILON (X)	Number that is almost negligible compared to one
HUGE (X)	Largest number in the model
MAXEXPONENT (X)	Maximum exponent in the model
MINEXPONENT (X)	Minimum exponent in the model
RADIX (X)	Base of the model
TINY (X)	Smallest number in the model

13.9.7 Transfer Function.

TRANSFER (SOURCE, MOLD, SIZE) Optional SIZE	Treat first argument as if of type of second argument

13.9.8 Floating-point Manipulation Functions.

EXPONENT (X)	Exponent part of a model number
FRACTION (X)	Fractional part of a number
NEAREST (X, S)	Nearest different processor number in given direction
RRSPACING (X)	Reciprocal of the relative spacing of model numbers near given number
SCALE (X, I)	Multiply a real by its base to an integer power
SETEXPONENT (X, I)	Set exponent part of a number
SPACING (X)	Absolute spacing of model numbers near given number

13.9.9 Vector and Matrix Multiply Functions.

DOTPRODUCT (VECTOR__A, VECTOR__B)	Dot product of two rank-one arrays
MATMUL (MATRIX__A, MATRIX__B)	Matrix multiplication

13.9.10 Array Reduction Functions.

ALL (MASK, DIM) Optional DIM	True if all values are true
ANY (MASK, DIM) Optional DIM	True if any value is true
COUNT (MASK, DIM) Optional DIM	Number of true elements in an array
MAXVAL (ARRAY, DIM, MASK) Optional DIM, MASK	Maximum value in an array
MINVAL (ARRAY, DIM, MASK) Optional DIM, MASK	Minimum value in an array
PRODUCT (ARRAY, DIM, MASK) Optional DIM, MASK	Product of array elements
SUM (ARRAY, DIM, MASK) Optional DIM, MASK	Sum of array elements

13.9.11 Array Inquiry Functions.

ALLOCATED (ARRAY)	Array allocation status
DLBOUND (ARRAY, DIM) Optional DIM	Declared lower dimension bounds of an array
DSHAPE (SOURCE)	Declared shape of an array or scalar

DSIZE (ARRAY, DIM) Optional DIM	Total declared number of elements in an array
DUBOUND (ARRAY, DIM) Optional DIM	Declared upper dimension bounds of an array
ELBOUND (ARRAY, DIM) Optional DIM	Effective lower dimension bounds of an array
ESHAPE (SOURCE)	Effective shape of an array or scalar
ESIZE (ARRAY, DIM) Optional DIM	Total effective number of elements in an array
EUBOUND (ARRAY, DIM) Optional DIM	Effective upper dimension bounds of an array

13.9.12 Array Construction Functions.

MERGE (TSOURCE, FSOURCE, MASK)	Merge under mask
PACK (ARRAY, MASK, VECTOR) Optional VECTOR	Pack an array into an array of rank one under a mask
RESHAPE (MOLD, SOURCE, PAD, ORDER) Optional PAD, ORDER	Reshape an array
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

13.9.13 Array Manipulation Functions.

CSHIFT (ARRAY, DIM, SHIFT)	Circular shift
EOSHIFT (ARRAY, DIM, SHIFT, BOUNDARY) Optional BOUNDARY	End-off shift
TRANSPOSE (MATRIX)	Transpose of an array of rank two

13.9.14 Array Geometric Location Functions.

MAXLOC (ARRAY,MASK) Optional MASK	Location of a maximum value in an array
MINLOC (ARRAY,MASK) Optional MASK	Location of a minimum value in an array

13.10 Table of Intrinsic Subroutines.

DATE_AND_TIME (ALL, COUNT, MSECOND, SECOND, MINUTE, HOUR, DAY, MONTH, YEAR, ZONE) Optional ALL, COUNT, MSECOND, SECOND, MINUTE, HOUR, DAY, MONTH, YEAR, ZONE	Obtain date and time
RANDOM (HARVEST)	Returns pseudorandom number
RANDOMSEED (SIZE, GET, PUT) Optional SIZE, GET, PUT	Initializes or restarts pseudorandom number generator
SYSTEM_CLOCK (COUNT, COUNT_RATE, COUNT_MAX) Optional COUNT, COUNT_RATE, COUNT_MAX	Obtain data from the system clock

13.11 Table of 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,...) Optional A3,...	REAL (MAX (A1, A2,A3,...)) Optional A3,...	integer
•	AMAX1 (A1,A2,A3,...) Optional A3,...	MAX (A1, A2,A3,...) Optional A3,...	default real
•	AMIN0 (A1,A2,A3,...) Optional A3,...	REAL (MIN (A1, A2,A3,...)) Optional A3,...	integer
•	AMIN1 (A1,A2,A3,...) Optional A3,...	MIN (A1, A2,A3,...) Optional A3,...	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)	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,...) Optional A3,...	MAX (A1,A2,A3,...) Optional A3,...	double precision real
•	DMIN1 (A1,A2,A3,...) Optional A3,...	MIN (A1,A2,A3,...) Optional 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)	integer
	IABS (A)	ABS (A)	integer
•	ICHAR (C)	ICHAR (C)	character
	IDIM (X,Y)	DIM (X,Y)	integer
•	IDINT (A)	INT (A)	double precision real
	IDNINT (A)	NINT (A)	double precision real
•	IFIX (A)	INT (A)	default real
	INDEX (S,T)	INDEX (S,T)	character
•	INT (A)	INT (A)	default real
	ISIGN (A,B)	SIGN (A,B)	integer
	LEN (S)	LEN (S)	character
•	LGE (S,T)	LGE (S,T)	character
•	LGT (S,T)	LGT (S,T)	character
•	LLE (S,T)	LLE (S,T)	character
•	LLT (S,T)	LLT (S,T)	character
•	MAX0 (A1,A2,A3,...) Optional A3,...	MAX (A1,A2,A3,...) Optional A3,...	integer
•	MAX1 (A1,A2,A3,...) Optional A3,...	INT (MAX (A1,A2,A3,...)) Optional A3,...	default real
•	MIN0 (A1,A2,A3,...) Optional A3,...	MIN (A1,A2,A3,...) Optional A3,...	integer
•	MIN1 (A1,A2,A3,...) Optional A3,...	INT (MIN (A1,A2,A3,...)) Optional A3,...	default real
	MOD (A,P)	MOD (A,P)	integer
	NINT (A)	NINT (A)	default real
•	REAL (A)	REAL (A)	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 function names must not be used as an actual argument.

13.12 Specifications of the Intrinsic Procedures. This section contains detailed specifications of all the intrinsic procedures in alphabetical order.

13.12.1 ABS (A).

Description. Absolute value.

Kind. Elemental function.

Argument. A must be of type integer, real, or complex.

Result Type and Type Parameters. The same as A except that if A is complex, the result is real.

Result Value. If A is of type integer 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).

13.12.2 ACHAR (I).

Description. Returns the character in a specified position of the ASCII collating sequence. It is the inverse of the IACHAR function.

Kind. Elemental function.

Argument. I must be of type integer.

Result Type and Type Parameters. Character of length one.

Result Value. If I has value in the range $0 \leq I \leq 127$, the result is the character in position I of the ASCII collating sequence; otherwise, the result is processor dependent. If the processor is not capable of representing both upper and lower case letters and I corresponds to an ASCII letter in a case that the processor is not capable of representing, the result is the letter in the case that the processor is capable of representing. ACHAR (IACHAR (C)) must have the value C for any character **C** capable of representation in the processor.

Example. ACHAR (88) has the value 'X'.

13.12.3 ACOS (X).

Description. Arccosine (inverse cosine) function.

Kind. Elemental function.

Argument. X must be of type real with a value that satisfies the inequality $|X| \leq 1$.

Result Type and Type Parameters. Same as X.

Result Value. The result has value equal to a processor-dependent approximation to arccos(X), expressed in radians. It lies in the range $0 \leq$ ACOS (X) $\leq \pi$.

Example. ACOS (0.54030231) has the value 1.0 (approximately).

13.12.4 ADJUSTL (STRING).

Description. Adjust to the left, removing leading blanks and inserting trailing blanks.

Kind. Elemental function.

Argument. STRING must be of type character.

Result Type and Type Parameters. Character of the same length as STRING.

Result Value. The value of the result is the same as STRING except that any leading blanks have been deleted and the same number of trailing blanks have been inserted.

Example. ADJUSTL (' WORD') has value 'WORD '.

13.12.5 ADJUSTR (STRING).

Description. Adjust to the right, removing trailing blanks and inserting leading blanks.

Kind. Elemental function.

Argument. STRING must be of type character.

Result Type and Type Parameters. Character of the same length as STRING.

Result Value. The value of the result is the same as STRING except that any trailing blanks have been deleted and the same number of leading blanks have been inserted.

Example. ADJUSTR ('WORD ') has value ' WORD'.

13.12.6. AIMAG (Z).

Description. Imaginary part of a complex number.

Kind. Elemental function.

Argument. Z must be of type complex.

Result Type and Type Parameters. Real with the same type parameters as Z.

Result Value. If Z has the value (x, y), the result has value y.

Example. AIMAG ((2.0, 3.0)) has the value 3.0.

13.12.7. AINT (A).

Description. Truncation to a whole number.

Kind. Elemental function.

Argument. A must be of type real.

Result Type and Type Parameters. Same as A.

Result Value. If $|A| < 1$, AINT (A) has the value 0; if $|A| \geq 1$, AINT (A) has value equal to the largest integer that does not exceed the magnitude of A and whose sign is the same as the sign of A.

Example. AINT (2.783) has the value 2.0.

13.12.8. ALL (MASK, DIM).

Optional Argument. DIM

Description. Determine whether all values are true in MASK along dimension DIM.

Kind. Transformational function.

Arguments.

MASK must be of type logical. It must not be scalar.

DIM (optional) must be scalar and of type integer with value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of MASK.

Result Type and Shape. The result is of type logical. It is scalar if DIM is absent or MASK has rank one; otherwise, the result is an array of rank $n-1$ and of shape $(d_1, d_2, \ldots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \ldots, d_n)$ where $(d_1, d_2, \ldots, d_n)$ is the shape of MASK.

Result Value.

Case (i): The result of ALL (MASK) has 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 value equal to that of ALL (MASK). Otherwise, the value of element $(s_1, s_2, \ldots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \ldots, s_n)$ of ALL (MASK, DIM) is equal to ALL (MASK $(s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)$).

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.].

13.12.9. ALLOCATED (ARRAY).

Description. Indicate whether or not an allocatable array is currently allocated.

Kind. Inquiry function.

Argument. ARRAY must be an allocatable array.

Result Type and Shape. The result is a logical scalar.

Result Value. The result has the value .TRUE. if ARRAY is currently allocated and has the value .FALSE. otherwise.

13.12.10. ANINT (A).

Description. Nearest whole number.

Kind. Elemental function.

Argument. A must be of type real.

Result Type and Type Parameters. Same as A.

Result Value. If $A > 0$, ANINT (A) has the value AINT (A + 0.5); if $A \leq 0$, ANINT (A) has the value AINT (A − 0.5).

Example. ANINT (2.783) has the value 3.0

13.12.11. ANY (MASK, DIM).

Optional Argument. DIM

Description. Determine whether any value is true in MASK along dimension DIM.

Kind. Transformational function.

Arguments.

MASK must be of type logical. It must not be scalar.

DIM (optional) must be scalar and of type integer with value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of MASK.

Result Type and Shape. The result is of type logical. It is scalar if DIM is absent or MASK has rank one; otherwise, the result is an array of rank $n-1$ and of shape $(d_1, d_2, ..., d_{\text{DIM}-1}, d_{\text{DIM}+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of MASK.

Result Value.

Case (i): The result of ANY (MASK) has value .TRUE. if any element of MASK is true and has value .FALSE. if no elements are true or if MASK has size zero.

Case (ii): If MASK has rank one, ANY (MASK, DIM) has value equal to that of ANY (MASK). Otherwise, the value of element $(s_1, s_2, ..., s_{\text{DIM}-1}, s_{\text{DIM}+1}, ..., s_n)$ of ANY (MASK, DIM) is equal to ANY (MASK $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$.

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.].

13.12.12. ASIN (X).

Description. Arcsine (inverse sine) function.

Kind. Elemental function.

Argument. X must be of type real. Its value must satisfy the inequality $|X| \leq 1$.

Result Type and Type Parameters. Same as X.

Result Value. The result has value equal to a processor-dependent approximation to arcsin(X), expressed in radians. It lies in the range $-\pi/2 \leq \text{ASIN}(X) \leq \pi/2$.

Example. ASIN (0.84147098) has the value 1.0 (approximately).

13.12.13. ATAN (X).

Description. Arctangent (inverse tangent) function.

Kind. Elemental function.

Argument. X must be of type real.

Result Type and Type Parameters. Same as X.

Result Value. The result has the value equal to a processor-dependent approximation to arctan(X), expressed in radians, that lies in the range $-\pi/2 \leq \text{ATAN}(X) \leq \pi/2$.

Example. ATAN (1.5574077) has the value 1.0 (approximately).

13.12.14. ATAN2 (Y, X).

Description. Arctangent (inverse tangent) function. The result is the principal value of the argument of the nonzero complex number (X, Y).

Kind. Elemental function.

Arguments.

Y	must be of type real.
X	must be of the same type as Y. If Y has value zero, X must not have value zero.

Result Type and Type Parameters. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to the argument of the complex number (X, Y), expressed in radians. It lies in the range $-\pi < \text{ATAN2}(Y, X) \leq \pi$ and is equal to a processor-dependent approximation to a value of arctan(Y/X) if $X \neq 0$. If $Y > 0$, the result is positive. If $Y = 0$, the result is zero if $X > 0$ and the result is π if $X < 0$. If $Y < 0$, the result is negative. If $X = 0$, the absolute value of the result is $\pi/2$.

Example. ATAN2 (1.5574077, 1.0) has the value 1.0 (approximately).

13.12.15. CHAR (I).

Description. Returns the character in a given position of the processor collating sequence. It is the inverse of the function ICHAR.

Kind. Elemental function.

Argument. I must be of type integer with a value in the range $0 \leq I \leq n-1$, where n is the number of characters in the collating sequence.

Result Type and Type Parameters. Character of length one.

Result Value. The result is the character in position I of the processor collating sequence. ICHAR (CHAR (I)) must have the value I for $0 \leq I \leq n-1$ and CHAR (ICHAR

(C)) must have the value C for any character C capable of representation in the processor.

Example. CHAR (88) has the value 'X' on a processor using the ASCII collating sequence.

13.12.16 CMPLX (X, Y, MOLD).

Optional Arguments. Y, MOLD

Description. Convert to complex type.

Kind. Elemental function.

Arguments.

X	must be of type integer, real, or complex.
Y (optional)	must be of type integer or real. It must not be present if X is of type complex.
MOLD (optional)	must be of type real.

Result Type and Type Parameters. The result is of type complex. If MOLD is present, the type parameters are those of MOLD; otherwise, the type parameters are those of default real type.

Result Value. If Y is absent and X is not complex, it is as if Y were present with the value zero. If X is complex, it is as if Y were present with the value AIMAG (X). If MOLD is absent, it is as if MOLD were present with default real type. CMPLX (X, Y, MOLD) has the complex value whose real part is REAL (X, MOLD) and whose imaginary part is REAL (Y, MOLD).

Example. CMPLX (– 3) has the value (– 3.0, 0.0).

13.12.17 CONJG (Z).

Description. Conjugate of a complex number.

Kind. Elemental function.

Argument. Z must be of type complex.

Result Type and Type Parameters. Same as Z.

Result Value. If Z has the value (x, y), the result has value (x, $-y$).

Example. CONJG ((2.0, 3.0)) has the value (2.0, – 3.0).

13.12.18 COS (X).

Description. Cosine function.

Kind. Elemental function.

Argument. X must be of type real or complex.

Result Type and Type Parameters. Same as X.

Result Value. The result has value equal to a processor-dependent approximation to cos(X). If X is of type real, 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).

13.12.19 COSH (X).

Description. Hyperbolic cosine function.

Kind. Elemental function.

Argument. X must be of type real.

Result Type and Type Parameters. Same as X.

Result Value. The result has value equal to a processor-dependent approximation to cosh(X).

Example. COSH (1.0) has the value 1.5430806 (approximately).

13.12.20 COUNT (MASK, DIM).

Optional Argument. DIM

Description. Count the number of true elements of MASK along dimension DIM.

Kind. Transformational function.

Arguments.

MASK	must be of type logical. It must not be scalar.
DIM (optional)	must be scalar and of type integer with value in the range $1 \le \text{DIM} \le n$, where n is the rank of MASK.

Result Type and Shape. The result is of type integer. It is scalar if DIM is absent or MASK has rank one; otherwise, the result is an array of rank $n-1$ and of shape $(d_1, d_2, \ldots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \ldots, d_n)$ where $(d_1, d_2, \ldots, d_n)$ is the shape of MASK.

Result Value.

Case (i): The result of COUNT (MASK) has value equal to the number of true elements of MASK or has value zero if MASK has size zero.

Case (ii): If MASK has rank one, COUNT (MASK, DIM) has value equal to that of COUNT (MASK). Otherwise, the value of element $(s_1, s_2, \ldots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \ldots, s_n)$ of COUNT (MASK, DIM) is equal to COUNT (MASK $(s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)$).

Examples.

Case (i): The value of COUNT ([.TRUE., .FALSE., .TRUE.]) is 2.

Case (ii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ and C is the array $\begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$, COUNT (B .NE. C, DIM = 1) is [2, 0, 1] and COUNT (B .NE. C, DIM = 2) is [1, 2].

13.12.21 CSHIFT (ARRAY, DIM, SHIFT).

Description. Perform a circular shift on an array expression of rank one or perform circular shifts on all the complete rank one sections along a given dimension of 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.

Kind. Transformational function.

Arguments.

ARRAY	may be of any type. It must not be scalar.
DIM	must be a scalar and of type integer with value in the range $1 \le \text{DIM} \le n$, where n is the rank of ARRAY.

SHIFT must be of type integer and must be scalar if ARRAY has rank one; otherwise, it must be scalar or of rank $n-1$ and of shape [E (1:DIM-1), E (DIM+1:n)] where E (1:n) is the shape of ARRAY.

Result Type, Type Parameters, and Shape. The result is of the type and type parameters of ARRAY, and has the shape of ARRAY.

Result Value.

Case (i): If ARRAY has rank one, the result is obtained by applying |SHIFT| circular shifts to ARRAY in the direction indicated by the sign of SHIFT. If SHIFT has value 1, element i of the result is ARRAY ($i+1$) for $i = 1, 2, ..., m-1$ and element m of the result is ARRAY (1) where m is the size of ARRAY. If SHIFT is positive, the result is equivalent to SHIFT applications of CSHIFT with SHIFT = 1. If SHIFT has value -1, element i of the result is ARRAY ($i-1$) for $i = 2, 3, ..., m$ and element 1 of the result is ARRAY (m). If SHIFT is negative, the result is equivalent to $-$SHIFT applications of CSHIFT with SHIFT = -1.

Case (ii): If ARRAY has rank greater than one, section (s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n) of the result has value equal to CSHIFT (ARRAY (s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n), 1, sh), where sh is SHIFT or SHIFT (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 circularly to the left by two positions is achieved by CSHIFT (V, DIM = 1, SHIFT = 2) which has the value [3, 4, 5, 6, 1, 2]; CSHIFT (V, DIM = 1, SHIFT = -2) achieves a circular shift to the right by two positions and has the value [5, 6, 1, 2, 3, 4].

Case (ii): The rows of an array of rank two may all be shifted by the same amount or by different amounts. If M is the array $\begin{bmatrix} A & B & C \\ A & B & C \\ A & B & C \end{bmatrix}$, the value of CSHIFT (M, DIM = 2, SHIFT = -1) is $\begin{bmatrix} C & A & B \\ C & A & B \\ C & A & B \end{bmatrix}$, and the value of CSHIFT (M, DIM = 2, SHIFT = [-1, 1, 0]) is $\begin{bmatrix} C & A & B \\ B & C & A \\ A & B & C \end{bmatrix}$.

13.12.22 DATE_AND_TIME (ALL, COUNT, MSECOND, SECOND, MINUTE, HOUR, DAY, MONTH, YEAR, ZONE).

Optional Arguments. ALL, COUNT, MSECOND, SECOND, MINUTE, HOUR, DAY, MONTH, YEAR, ZONE

Description. Returns integer data from the date available to the processor and a real-time clock.

Kind. Subroutine.

Arguments.

ALL (optional) must be of type integer and rank one. Its size must be at least 9. The values returned in ALL are as for the remaining 9 arguments, taken in order.

COUNT (optional) must be scalar and of type integer. It is set to a processor-dependent value based on the current value of the basic clock or to $-$HUGE (0) if there is no clock. The processor-dependent value is incremented by one for each clock count until the value COUNT_MAX (as returned by subroutine SYSTEM_CLOCK) is

reached and is reset to zero at the next count. It lies in the range 0 to COUNT_MAX if there is a clock.

MSECOND (optional) must be scalar and of type integer. It is set to the millisecond part of the local time, or to −HUGE (0) if there is no clock. It lies in the range 0 to 999 if there is a clock.

SECOND (optional) must be scalar and of type integer. It is set to the second part of the local time, or to −HUGE (0) if there is no clock. It lies in the range 0 to 59 if there is a clock.

MINUTE (optional) must be scalar and of type integer. It is set to the minute part of the local time, or to −HUGE (0) if there is no clock. It lies in the range 0 to 59 if there is a clock.

HOUR (optional) must be scalar and of type integer. It is set to the hour part of the local time, or to −HUGE (0) if there is no clock. It lies in the range 0 to 23 if there is a clock.

DAY (optional) must be scalar and of type integer. It is set to the day of the month, or to −HUGE (0) if there is no date available. It lies in the range 1 to 31 if there is a date available.

MONTH (optional) must be scalar and of type integer. It is set to the month of the year, or to −HUGE (0) if there is no date available. It lies in the range 1 to 12 if there is a date available.

YEAR (optional) must be scalar and of type integer. It is set to the year according to the Gregorian calendar (e.g. 1988), or to −HUGE (0) if there is no date available.

ZONE (optional) must be scalar and of type integer. It is set to the number of minutes that local time is in advance of Greenwich Mean Time, or to −HUGE (0) if this information is not available.

Example.

```
CALL DATE_AND_TIME (ZONE = HERE)
```

will assign the value −300 to the variable HERE if the local time is 5 hours behind GMT.

13.12.23 DBLE (A).

Description. Convert to double precision real type.

Kind. Elemental function.

Argument. A must be of type integer, real, or complex.

Result Type. Double precision real.

Result Value.

Case (i): If A is of type double precision, DBLE (A) = A.

Case (ii): If A is of type integer or real, the result is as much precision of the significant part of A as a double precision datum can contain.

Case (iii): If A is of type complex, the result is as much precision of the significant part of the real part of A as a double precision datum can contain.

Example. DBLE (−3) has the value −3.0D0.

13.12.24 DIGITS (X).

Description. Returns the number of significant digits in the model representing numbers of the same type and type parameters as the argument.

Kind. Inquiry function.

Argument. X must be of type integer or real. It may be scalar or array valued.

Result Type and Shape. Integer scalar.

Result Value. The result has value q if X is of type integer and p if X is of type real, where q and p are as defined in 13.6.1 for the model representing numbers of the same type and type parameters as X.

Example. DIGITS (X) has the value 24 for real X whose model is as at the end of 13.6.1.

13.12.25 DIM (X, Y).

Description. The difference X – Y if it is positive; otherwise zero.

Kind. Elemental function.

Arguments.

X	must be of type integer or real.
Y	must be of the same type as X.

Result Type and Type Parameters. Same as X.

Result Value. The value of the result is X – Y if X > Y and zero otherwise.

Example. DIM (–3.0, 2.0) has the value 0.0.

13.12.26 DLBOUND (ARRAY, DIM).

Optional Argument. DIM

Description. Returns all the declared lower bounds of an array or a specified declared lower bound.

Kind. Inquiry function.

Arguments.

ARRAY	may be of any type. It must not be scalar. It must not be an allocatable array that is not allocated or an alias array that is not alias associated.
DIM (optional)	must be scalar and of type integer with value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY.

Result Type and Shape. The result is of type integer. It is scalar if DIM is present; otherwise, the result is an array of rank one and size n, where n is the rank of ARRAY.

Result Value.

Case (i): DLBOUND (ARRAY, DIM) has value equal to the declared lower bound for subscript DIM of ARRAY if dimension DIM of ARRAY does not have size zero and has the value 1 if dimension DIM has size zero. For an array section or an array expression, it has the value 1.

Case (ii): DLBOUND (ARRAY) has value whose ith component is equal to DLBOUND (ARRAY, i), for $i = 1, 2, \ldots, n$, where n is the rank of ARRAY.

Example. If A is declared by the statement

```
REAL A (2:3, 7:10)
```

then DLBOUND (A) is [2, 7] and DLBOUND (A, DIM = 2) is 7.

13.12.27 DOTPRODUCT (VECTOR_A, VECTOR_B).

Description. Performs dot-product multiplication of numeric or logical vectors.

Kind. Transformational function.

Arguments.

VECTOR_A	must be of numeric type (integer, real, or complex) or of logical type. It must be array valued and of rank one.
VECTOR_B	must be of numeric type if VECTOR_A is of numeric type or of type logical if VECTOR_A is of type logical. It must be array valued and of rank one. It must be of the same size as VECTOR_A.

Result Type, Type Parameters, and Shape. If the arguments are of numeric type, the type and type parameters of the result are those of the expression VECTOR_A * VECTOR_B determined by the types of the arguments according to 7.1.4. If the arguments are of type logical, the result is of type logical. The result is scalar.

Result Value.

Case (i): If VECTOR_A is of type integer or real, the result has value SUM (VECTOR_A*VECTOR_B). If the vectors have size zero, the result has value zero.

Case (ii): If VECTOR_A is of type complex, the result has value SUM (CONJG (VECTOR_A)*VECTOR_B). If the vectors have size zero, the result has value zero.

Case (iii): If VECTOR_A is of type logical, the result has value ANY (VECTOR_A .AND. VECTOR_B). If the vectors have size zero, the result has value .FALSE.

Example. DOTPRODUCT ([1, 2, 3], [2, 3, 4]) has the value 20.

13.12.28 DPROD (X, Y).

Description. Double precision real product.

Kind. Elemental function.

Arguments.

X	must be of type default real.
Y	must be of type default real.

Result Type. Double precision real.

Result Value. The value of the result is X * Y.

Example. DPROD (−3.0, 2.0) has the value −6.0D0.

13.12.29 DSHAPE (SOURCE).

Description. Returns the declared shape of an array or a scalar.

Kind. Inquiry function.

Argument. SOURCE may be of any type. It may be array valued or scalar. It must not be an allocatable array that is not allocated or an alias array that is not alias associated. It must not be an assumed-size array.

Result Type and Shape. The result is an integer array of rank one whose size is equal to the rank of SOURCE.

Result Value. The value of the result is the declared shape of SOURCE.

Examples. The value of DSHAPE (A (2:5, −1:1)) is [4, 3]. The value of DSHAPE (3) is the rank-one array of size zero.

13.12.30 DSIZE (ARRAY, DIM).

Optional Argument. DIM

Description. Returns the declared extent of an array along a specified dimension or the total declared number of elements in the array.

Kind. Inquiry function.

Arguments.

ARRAY	may be of any type. It must not be scalar. If must not be an allocatable array that is not allocated or an alias array that is not alias associated. If ARRAY is an assumed-size array, DIM must be present with value less than the rank of ARRAY.
DIM (optional)	must be scalar and of type integer with value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY.

Result Type and Shape. Integer scalar.

Result Value. The result has value equal to the declared extent of dimension DIM of ARRAY or, if DIM is absent, the total declared number of elements of ARRAY.

Examples. The value of DSIZE (A (2:5, −1:1), DIM=2) is 3. The value of DSIZE (A (2:5, −1:1)) is 12.

13.12.31 DUBOUND (ARRAY, DIM).

Optional Argument. DIM

Description. Returns all the declared upper bounds of an array or a specified declared upper bound.

Kind. Inquiry function.

Arguments.

ARRAY	may be of any type. It must not be scalar. It must not be an allocatable array that is not allocated or an alias array that is not alias associated. If DIM is omitted or is present with value equal to the rank of ARRAY, ARRAY must not be an assumed-size array.
DIM (optional)	must be scalar and of type integer with value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY.

Result Type and Shape. The result is of type integer. It is scalar if DIM is present; otherwise, the result is an array of rank one and size n, where n is the rank of ARRAY.

Result Value.

Case (i): DUBOUND (ARRAY, DIM) has value equal to the declared upper bound for subscript DIM of ARRAY if dimension DIM of ARRAY does not have size zero and has the value zero if dimension DIM has size zero. For an array section or an array expression, its value is the number of elements in the corresponding dimension.

Case (ii): DUBOUND (ARRAY) has value whose ith component is equal to DUBOUND (ARRAY, i), for $i = 1, 2, \ldots, n$, where n is the rank of ARRAY.

Example. If A is declared by the statement

```
REAL A (2:3, 7:10)
```

then DUBOUND (A) is [3, 10] and DUBOUND (A, DIM=2) is 10.

13.12.32 EFFECTIVE_EXPONENT_RANGE (X).

Description. Returns the decimal exponent range in the model representing numbers of the same type and type parameters as the argument.

Kind. Inquiry function.

Argument. X must be of type real, complex, or derived type with type parameters named PRECISION and EXPONENT_RANGE. It may be scalar or array valued.

Result Type and Shape. Integer scalar.

Result Value. The result has value INT (MIN (LOG10 (*huge*), −LOG10 (*tiny*))), where *huge* and *tiny* are the largest and smallest numbers in the model representing real numbers with the same values for the type parameters as X (see 13.6.1).

Example. EFFECTIVE_EXPONENT_RANGE (X) has the value 38 for real X whose model is as at the end of 13.6.1, since in this case *huge* = $(1-2^{-24}) \times 2^{127}$ and *tiny* = 2^{-127}.

13.12.33 EFFECTIVE_PRECISION (X).

Description. Returns the decimal precision in the model representing numbers of the same type and type parameters as the argument.

Kind. Inquiry function.

Argument. X must be of type real, complex, or derived type with type parameters named PRECISION and EXPONENT_RANGE. It may be scalar or array valued.

Result Type and Shape. Integer scalar.

Result Value. The result has value INT (($p-1$) * LOG10 (b)) + k, where b and p are as defined in 13.6.1 for the model representing real numbers with the same values for the type parameters as X, and where k is 1 if b is an integral power of 10 and 0 otherwise.

Example. EFFECTIVE_PRECISION (X) has the value INT (23 * LOG10 (2.)) = INT (6.92...) = 6 for real X whose model is as at the end of 13.6.1.

13.12.34 ELBOUND (ARRAY, DIM).

Optional Argument. DIM

Description. Returns all the effective lower bounds of an array or a specified effective lower bound.

Kind. Inquiry function.

Arguments.

ARRAY	may be of any type. It must not be scalar. It must not be an allocatable array that is not allocated or an alias array that is not alias associated.
DIM (optional)	must be scalar and of type integer with value in the range $1 \le \text{DIM} \le n$, where n is the rank of ARRAY.

Result Type and Shape. The result is of type integer. It is scalar if DIM is present; otherwise, the result is an array of rank one and size n, where n is the rank of ARRAY.

Result Value.

Case (i): ELBOUND (ARRAY, DIM) has value equal to the effective lower bound for subscript DIM of ARRAY if dimension DIM of ARRAY does not have size zero and has the value 1 if dimension DIM has size zero. For an array section or an array expression, it has the value 1.

Case (ii): ELBOUND (ARRAY) has value whose *i*th component is equal to ELBOUND (ARRAY, *i*), for $i = 1, 2, ..., n$, where n is the rank of ARRAY.

Example. If A is declared and its range is set as follows:

```
REAL, RANGE :: A (2:10, 5:10)
SET RANGE (4:6, 7:9) A
```

then ELBOUND (A) is [4, 7] and ELBOUND (A, DIM=2) is 7.

13.12.35 EOSHIFT (ARRAY, DIM, SHIFT, BOUNDARY).

Optional Argument. BOUNDARY

Description. Perform an end-off shift on an array expression of rank one or perform end-off shifts on all the complete rank-one sections along a given dimension of 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.

Kind. Transformational function.

Arguments.

ARRAY may be of any type. It must not be scalar.

DIM must be scalar and of type integer with value in the range $1 \le \text{DIM} \le n$, where n is the rank of ARRAY.

SHIFT must be of type integer and must be scalar if ARRAY has rank one; otherwise, it must be scalar or of rank $n-1$ and of shape [E (1:DIM − 1), E (DIM + 1:*n*)], where E (1:*n*) is the shape of ARRAY.

BOUNDARY (optional)
must be of the same type and type parameters as ARRAY and must be scalar if ARRAY has rank one; otherwise, it must be either scalar or of rank $n-1$ and of shape [E (1:DIM-1), E (DIM + 1:*n*)]. BOUNDARY may be omitted for the data types in the following table and, in this case, it is as if it were present with the scalar value shown.

Type of ARRAY	Value of BOUNDARY
Integer	0
Real	0.0
Double precision	0.0D0
Complex	(0.0, 0.0)
Logical	.FALSE.
Character (*len*)	*len* blanks

Result Type, Type Parameters, and Shape. The result has the type, type parameters, and shape of ARRAY.

Result Value. Element $(s_1, s_2, ..., s_n)$ of the result has the value ARRAY $(s_1, s_2, ..., s_{\text{DIM}-1}, s_{\text{DIM}} + sh, s_{\text{DIM}+1}, ..., s_n)$ where *sh* is SHIFT or SHIFT $(s_1, s_2, ..., s_{\text{DIM}-1}, s_{\text{DIM}+1}, ..., s_n)$ provided the inequality ELBOUND (ARRAY, DIM) $\le s_{\text{DIM}} + sh \le$ EUBOUND (ARRAY, DIM) holds and is otherwise BOUNDARY or BOUNDARY $(s_1, s_2, ..., s_{\text{DIM}-1}, s_{\text{DIM}+1}, ..., s_n)$.

Examples.

Case (i): If V is the array [1, 2, 3, 4, 5, 6], the effect of shifting V end-off to the left by 3 positions is achieved by EOSHIFT (V, DIM = 1, SHIFT = 3) which has the value [4, 5, 6, 0, 0, 0]; EOSHIFT (V, DIM = 1, SHIFT = −2, BOUNDARY = 99) achieves an end-off shift to the right by 2 positions with the boundary value of 99 and has the value [99, 99, 1, 2, 3, 4].

Case (ii): The rows of an array of rank two may all be shifted by the same amount or by different amounts and the boundary elements can be the same or different. If M is the array $\begin{bmatrix} A & B & C \\ A & B & C \\ A & B & C \end{bmatrix}$, then the value of EOSHIFT (M, DIM = 2, SHIFT = −1, BOUNDARY = '*') is $\begin{bmatrix} * & A & B \\ * & A & B \\ * & A & B \end{bmatrix}$, and the value of EOSHIFT (M, DIM = 2, SHIFT = [−1, 1, 0], BOUNDARY = ['*', '/', '?']) is $\begin{bmatrix} * & A & B \\ B & C & / \\ A & B & C \end{bmatrix}$.

13.12.36 EPSILON (X).

Description. Returns a positive model number that is almost negligible compared to unity in the model representing numbers of the same type and type parameters as the argument.

Kind. Inquiry function.

Argument. X must be of type real. It may be scalar or array valued.

Result Type, Type Parameters, and Shape. Scalar of the same type and type parameters as X.

Result Value. The result has value b^{1-p} where b and p are as defined in 13.6.1 for the model representing numbers of the same type and type parameters as X.

Example. EPSILON (X) has the value 2^{-23} for real X whose model is as at the end of 13.6.1.

13.12.37 ESHAPE (SOURCE).

Description. Returns the effective shape of an array or a scalar.

Kind. Inquiry function.

Argument. SOURCE may be of any type. It may be array valued or scalar. It must not be an assumed-size array.

Result Type and Shape. The result is an integer array of rank one whose size is equal to the rank of SOURCE.

Result Value. The value of the result is the effective shape of SOURCE.

Examples.

The value of ESHAPE (A (2:5, −1:1)) is [4, 3].

The value of ESHAPE (3) is the rank-one array of size zero.

If B is declared and its range is set as follows:

```
INTEGER, RANGE, ARRAY (20, 300) :: B
SET RANGE (5:15, :) B
```

then ESHAPE (B) is [11, 300] and ESHAPE (B (1:4, 10)) is [4].

13.12.38 ESIZE (ARRAY, DIM).

Optional Argument. DIM

Description. Returns the effective extent of an array along a specified dimension or the total effective number of elements in the array.

Kind. Inquiry function.

Arguments.

ARRAY — may be of any type. It must not be scalar. If ARRAY is an assumed-size array, DIM must be present with value less than the rank of ARRAY.

DIM (optional) — must be scalar and of type integer with value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY.

Result Type and Shape. Integer scalar.

Result Value. The result has value equal to the effective extent of dimension DIM of ARRAY or, if DIM is absent, the total effective number of elements of ARRAY.

Examples.

The value of ESIZE (A (2:5, −1:1), DIM=2) is 3.

The value of ESIZE (A (2:5, −1:1)) is 12.

If B is declared and its range is set as follows:

```
INTEGER, RANGE, ARRAY (20, 300) :: B
SET RANGE (5:15, :) B
```

then ESIZE (B) is 3300 and ESIZE (B (1:4, 10), DIM = 1) is 4.

13.12.39 EUBOUND (ARRAY, DIM).

Optional Argument. DIM

Description. Returns all the effective upper bounds of an array or a specified effective upper bound.

Kind. Inquiry function.

Arguments.

ARRAY — may be of any type. It must not be scalar. It must not be an allocatable array that is not allocated or an alias array that is not associated. If DIM is omitted or is present with value equal to the rank of ARRAY, ARRAY must not be an assumed-size array.

DIM (optional) — must be scalar and of type integer with value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY.

Result Type and Shape. The result is of type integer. It is scalar if DIM is present; otherwise, the result is an array of rank one and size n, where n is the rank of ARRAY.

Result Value.

Case (i): EUBOUND (ARRAY, DIM) has value equal to the effective upper bound for subscript DIM of ARRAY if dimension DIM of ARRAY does not have size zero and has the value zero if dimension DIM has size zero. For an array section or an array expression, its value is the number of elements in the corresponding dimension.

Case (ii): EUBOUND (ARRAY) has value whose ith component is equal to EUBOUND (ARRAY, i), for $i = 1, 2, \ldots, n$, where n is the rank of ARRAY.

Examples. If A is declared by the statement

```
REAL A (2:3, 7:10)
```

then EUBOUND (A) is [3, 10] and EUBOUND (A, DIM=2) is 10.

If B is declared and its range is set as follows:

```
INTEGER, RANGE, ARRAY (20, 300) :: B
SET RANGE (5:15, :) B
```

then EUBOUND (B) is [15, 300] and EUBOUND (B, DIM = 1) is 15. Furthermore, EUBOUND (B (2:5, 10)) is [4].

13.12.40 EXP (X).

Description. Exponential.

Kind. Elemental function.

Argument. X must be of type real or complex.

Result Type and Type Parameters. Same as X.

Result Value. The result has value equal to a processor-dependent approximation to e^X. If X is of type complex, its imaginary part is regarded as a value in radians.

Example. EXP (1.0) has the value 2.7182818 (approximately).

13.12.41 EXPONENT (X).

Description. Returns the exponent part of the argument when represented as a model number.

Kind. Elemental function.

Argument. X must be of type real.

Result Type. Integer.

Result Value. The result has value equal to the exponent e of the model representation (see 13.6.1) for the value of X, provided X is nonzero and e is within range for integers. EXPONENT (X) has 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 13.6.1.

13.12.42 FRACTION (X).

Description. Returns the fractional part of the model representation of the argument value.

Kind. Elemental function.

Argument. X must be of type real.

Result Type and Type Parameters. Same as X.

Result Value. The result has value $X \times b^{-e}$, where b and e are as defined in 13.6.1 for the model representation of X. If X has value zero, the result has value zero.

Example. FRACTION (3.0) has the value 0.75 for reals whose model is as at the end of 13.6.1.

13.12.43 HUGE (X).

Description. Returns the largest number in the model representing numbers of the same type and type parameters as the argument.

Kind. Inquiry function.

Argument. X must be of type integer or real. It may be scalar or array valued.

Result Type, Type Parameters, and Shape. Scalar of the same type and type parameters as X.

Result Value. The result has value $r^q - 1$ if X is of type integer and $(1 - b^{-p})b^{e_{max}}$ if X is of type real, where r, q, b, p, and e_{max} are as defined in 13.6.1 for the model representing numbers of the same type and type parameters as X.

Example. HUGE (X) has the value $(1 - 2^{-24}) \times 2^{127}$ for real X whose model is as at the end of 13.6.1.

13.12.44 IACHAR (C).

Description. Returns the position of a character in the ASCII collating sequence.

Kind. Elemental function.

Argument. C must be of type character and of length one.

Result Type. Integer.

Result Value. If C is in the collating sequence described in ANSI X3.4-1977 (ASCII), the result is the position of C in that sequence and satisfies the inequality (0 ≤ IACHAR (C) ≤ 127). A processor-dependent value is returned if C is not in the ASCII collating sequence. The results must be consistent with the LGE, LGT, LLE, and LLT lexical comparison functions. For example, if LLE (C, D) is true, IACHAR (C) .LE. IACHAR (D) is true where C and D are any two characters representable by the processor.

Example. IACHAR ('X') has the value 88.

13.12.45 ICHAR (C).

Description. Returns the position of a character in the processor collating sequence.

Kind. Elemental function.

Argument. C must be of type character and of length one. Its value must be that of a character capable of representation in the processor.

Result Type. Integer.

Result Value. The result is the position of C in the processor collating sequence and is in the range $0 \le$ ICHAR (C) $\le n - 1$, where n is the number of characters in the collating sequence. For any characters C and D capable of representation in the processor, C .LE. D is true if and only if ICHAR (C) .LE. ICHAR (D) is true and C .EQ. D is true if and only if ICHAR (C). EQ. ICHAR (D) is true.

Example. ICHAR ('X') has the value 88 on a processor using the ASCII collating sequence.

13.12.46 INDEX (STRING, SUBSTRING, BACK).

Optional Argument. BACK

Description. Returns the starting position of a substring within a string.

Kind. Elemental function.

Arguments.

STRING	must be of type character.
SUBSTRING	must be of type character.

BACK (optional) must be of type logical.

Result Type. Integer.

Result Value.

Case (i): If BACK is absent or present with the value .FALSE., the value returned is the minimum 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 value returned is the maximum 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 LEN (STRING) + 1 is returned if LEN (SUBSTRING) = 0.

Examples. INDEX ('FORTRAN', 'R') has value 3.
INDEX ('FORTRAN', 'R', BACK = .TRUE.) has the value 5.

13.12.47 INT (A).

Description. Convert to integer type.

Kind. Elemental function.

Argument. A must be of type integer, real, or complex.

Result Type. Integer.

Result Value.

Case (i): If A is of type integer, INT (A) = A.

Case (ii): If A is of type real, there are two cases: if |A| < 1, INT (A) has the value 0; if |A| ≥ 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.

13.12.48 LEN (STRING).

Description. Returns the length of a character entity.

Kind. Inquiry function.

Argument. STRING must be of type character. It may be scalar or array valued.

Result Type and Shape. Integer scalar.

Result Value. The result has value equal to the number of characters in STRING if it is scalar or in an element of STRING if it is array valued.

Example. If C is declared by the statement

```
CHARACTER (11) C (100)
```

LEN (C) has value 11.

13.12.49 LEN_TRIM (STRING).

Description. Returns the length of the character argument without trailing blank characters.

Kind. Elemental function.

Argument. STRING must be of type character.

Result Type. Integer.

Result Value. The result has a value equal to the number of characters 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 value 4 and LEN_TRIM (' ') has value 0.

13.12.50 LGE (STRING_A, STRING_B).

Description. Test whether a string is lexically greater than or equal to another string, based on the ASCII collating sequence.

Kind. Elemental function.

Arguments.

STRING_A must be of type character.

STRING_B must be of type character.

Result Type. Logical.

Result Value. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string. If either string contains a character not in the ASCII character set, the result is processor dependent. The result is true if the strings are equal or if STRING_A follows STRING_B in the collating sequence described in ANSI X3.4-1977 (ASCII); otherwise, the result is false.

Example. LGE ('ONE', 'TWO') has the value .FALSE.

13.12.51 LGT (STRING_A, STRING_B).

Description. Test whether a string is lexically greater than another string, based on the ASCII collating sequence.

Kind. Elemental function.

Arguments.

STRING_A must be of type character.

STRING_B must be of type character.

Result Type. Logical.

Result Value. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string. If either string contains a character not in the ASCII character set, the result is processor dependent. The result is true if STRING_A follows STRING_B in the collating sequence described in ANSI X3.4-1977 (ASCII); otherwise, the result is false.

Example. LGT ('ONE', 'TWO') has the value .FALSE.

13.12.52 LLE (STRING_A, STRING_B).

Description. Test whether a string is lexically less than or equal to another string, based on the ASCII collating sequence.

Kind. Elemental function.

Arguments.

STRING_A must be of type character.

STRING_B must be of type character.

Result Type. Logical.

Result Value. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string. If either string contains a character not in the ASCII character set, the result is processor dependent. The result is true if the strings are equal or if STRING_A precedes STRING_B in the collating sequence described in ANSI X3.4-1977 (ASCII); otherwise, the result is false.

Example. LLE ('ONE', 'TWO') has the value .TRUE.

13.12.53 LLT (STRING_A, STRING_B).

Description. Test whether a string is lexically less than another string, based on the ASCII collating sequence.

Kind. Elemental function.

Arguments.

STRING_A must be of type character.

STRING_B must be of type character.

Result Type. Logical.

Result Value. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string. If either string contains a character not in the ASCII character set, the result is processor dependent. The result is true if STRING_A precedes STRING_B in the collating sequence described in ANSI X3.4-1977 (ASCII); otherwise, the result is false.

Example. LLT ('ONE', 'TWO') has the value .TRUE.

13.12.54 LOG (X).

Description. Natural logarithm.

Kind. Elemental function.

Argument. X must be of type real or complex. If X is real, its value must be greater than zero. If X is complex, its value must not be zero.

Result Type and Type Parameters. Same as X.

Result Value. The result has value equal to a processor-dependent approximation to $\log_e X$. A result of type complex is the principal value with imaginary part ω in the range $-\pi < \omega \leq \pi$. The imaginary part of the result is π only when the real part of the argument is less than zero and the imaginary part of the argument is zero.

Example. LOG (10.0) has the value 2.3025851 (approximately).

13.12.55 LOG10 (X).

Description. Common logarithm.

Kind. Elemental function.

Argument. X must be of type real. The value of X must be greater than zero.

Result Type and Type Parameters. Same as X.

Result Value. The result has value equal to a processor-dependent approximation to $\log_{10} X$.

Example. LOG10 (10.0) has the value 1.0 (approximately).

13.12.56 MATMUL (MATRIX_A, MATRIX_B).

Description. Performs matrix multiplication of numeric or logical matrices.

Kind. Transformational function.

Arguments.

MATRIX_A must be of numeric type (integer, real, or complex) or of logical type. It must be array valued and of rank one or two. Its shape must be defined.

MATRIX_B must be of numeric type if MATRIX_A is of numeric type and of logical type if MATRIX_A is of logical type. It must be array valued and of rank one or two. If MATRIX_A has rank one, MATRIX_B must have rank two. Its shape must be defined. The size of the first (or only) dimension of MATRIX_B must equal the size of the last (or only) dimension of MATRIX_A.

Result Type, Type Parameters, and Shape. If the arguments are of numeric type, the type and type parameters of the result are determined by the types of the arguments according to 7.1.4. If the arguments are of type logical, the result is of type logical. The shape of the result depends on the shapes of the arguments as follows:

Case (i): If MATRIX_A has shape [*n*, *m*] and MATRIX_B has shape [*m*, *k*], the result has shape [*n*, *k*].

Case (ii): If MATRIX_A has shape [*m*] and MATRIX_B has shape [*m*, *k*], the result has shape [*k*].

Case (iii): If MATRIX_A has shape [*n*, *m*] and MATRIX_B has shape [*m*], the result has shape [*n*].

Result Value.

Case (i): Element (*i*, *j*) of the result has value SUM (MATRIX_A (*i*, :) * MATRIX_B (:, *j*)) if the arguments are of numeric type and has value ANY (MATRIX_A (*i*, :) .AND. MATRIX_B (:, *j*)) if the arguments are of logical type.

Case (ii): Element (*j*) of the result has value SUM (MATRIX_A (:) * MATRIX_B (:, *j*)) if the arguments are of numeric type and has value ANY (MATRIX_A (:) .AND. MATRIX_B (:, *j*)) if the arguments are of logical type.

Case (iii): Element (*i*) of the result has value SUM (MATRIX_A (*i*, :) * MATRIX_B (:)) if the arguments are of numeric type and has value ANY (MATRIX_A (*i*, :) .AND. MATRIX_B (:)) if the arguments are of logical type.

Examples. Let A and B be the matrices $\begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \end{bmatrix}$ and $\begin{bmatrix} 1 & 2 \\ 2 & 3 \\ 3 & 4 \end{bmatrix}$; let X and Y be the vectors [1, 2] and [1, 2, 3].

Case (i): The result of MATMUL (A, B) is the matrix-matrix product AB with value $\begin{bmatrix} 14 & 20 \\ 20 & 29 \end{bmatrix}$.

Case (ii): The result of MATMUL (X, A) is the vector-matrix product XA with value [5, 8, 11].

Case (iii): The result of MATMUL (A, Y) is the matrix-vector product AY with value [14, 20].

13.12.57 MAX (A1, A2, A3, ...).

Optional Arguments. A3, ...

Description. Maximum value.

Kind. Elemental function.

Arguments. The arguments must all have the same type which must be integer or real and they must all have the same type parameters.

Result Type and Type Parameters. Same as the arguments.

Result Value. The value of the result is that of the largest argument.

Example. MAX (−9.0, 7.0, 2.0) has the value 7.0.

13.12.58 MAXEXPONENT (X).

Description. Returns the maximum exponent in the model representing numbers of the same type and type parameters as the argument.

Kind. Inquiry function.

Argument. X must be of type real. It may be scalar or array valued.

Result Type and Shape. Integer scalar.

Result Value. The result has value e_{max}, as defined in 13.6.1 for the model representing numbers of the same type and type parameters as X.

Example. MAXEXPONENT (X) has the value 127 for real X whose model is as at the end of 13.6.1.

13.12.59 MAXLOC (ARRAY, MASK).

Optional Argument. MASK

Description. Determine the location of an element of ARRAY having the maximum value of the elements identified by MASK.

Kind. Transformational function.

Arguments.

ARRAY	must be of type integer or real. It must not be scalar.
MASK (optional)	must be of type logical and must be conformable with ARRAY.

Result Type and Shape. The result is of type integer; it is an array of rank one and of size equal to the rank of ARRAY.

Result Value.

Case (i): If MASK is absent, the result is a rank-one array whose element values are the values of the subscripts of an element of ARRAY whose value equals the maximum value of all of the elements of ARRAY. The *i*th subscript returned lies in the range 1 to e_i, where e_i is the extent of the *i*th dimension of ARRAY. If more than one element has maximum value, the element whose subscripts are returned is processor dependent. If ARRAY has size zero, the value of the result is processor dependent.

Case (ii): If MASK is present, the result is a rank-one array whose element values are the values of the subscripts of an element of ARRAY, corresponding to a true element of MASK, whose value equals the maximum value of all such elements of ARRAY. The *i*th subscript returned lies in the range 1 to e_i, where e_i is the extent of the *i*th dimension of ARRAY. If more than one such element has maximum value, the element whose subscripts are returned is processor dependent. If there are no such elements (that is, if

ARRAY has size zero or every element of MASK has the value .FALSE.), the value of the result is processor dependent.

Examples.

Case (i): The value of MAXLOC ([2, 4, 6]) is [3].

Case (ii): If A has the value $\begin{bmatrix} 0 & -5 & 8 & -3 \\ 3 & 4 & -1 & 2 \\ 1 & 5 & 6 & -4 \end{bmatrix}$, MAXLOC (A, MASK = A.LT.6) has the value [3, 2].

13.12.60 MAXVAL (ARRAY, DIM, MASK).

Optional Arguments. DIM, MASK

Description. Maximum value of the elements of ARRAY along dimension DIM corresponding to the true elements of MASK.

Kind. Transformational function.

Arguments.

ARRAY	must be of type integer or real. It must not be scalar. Its shape must be defined.
DIM (optional)	must be scalar and of type integer with value in the range $1 \le \text{DIM} \le n$, where n is the rank of ARRAY.
MASK (optional)	must be of type logical and must be conformable with ARRAY.

Result Type, Type Parameters, and Shape. The result is of the same type and type parameters as ARRAY. It is scalar if DIM is absent or ARRAY has rank one; otherwise, the result is an array of rank $n-1$ and of shape $(d_1, d_2, \ldots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \ldots, d_n)$ where $(d_1, d_2, \ldots, d_n)$ is the shape of ARRAY.

Result Value.

Case (i): The result of MAXVAL (ARRAY) has value equal to the maximum value of all the elements of ARRAY or has value −HUGE (ARRAY) if ARRAY has size zero.

Case (ii): The result of MAXVAL (ARRAY, MASK = MASK) has value equal to the maximum value of the elements of ARRAY corresponding to true elements of MASK or has value −HUGE (ARRAY) if there are no true elements.

Case (iii): If ARRAY has rank one, MAXVAL (ARRAY, DIM [,MASK]) has value equal to that of MAXVAL (ARRAY [,MASK = MASK]). Otherwise, the value of element $(s_1, s_2, \ldots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \ldots, s_n)$ of MAXVAL (ARRAY, DIM [,MASK]) is equal to MAXVAL (ARRAY $(s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)$, [, MASK = MASK $(s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)$]).

Examples.

Case (i): The value of MAXVAL ([1, 2, 3]) is 3.

Case (ii): MAXVAL (C, MASK = C .GT. 0.0) finds the maximum of the positive elements of C.

Case (iii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$, MAXVAL (B, DIM = 1) is [2, 4, 6] and MAXVAL (B, DIM = 2) is [5, 6].

13.12.61 MERGE (TSOURCE, FSOURCE, MASK).

Description. Choose alternative value according to value of a mask.

Kind. Elemental function.

Arguments.

TSOURCE	may be of any type.
FSOURCE	must be of the same type and type parameters as TSOURCE.
MASK	must be of type logical.

Result Type and Type Parameters. Same as TSOURCE.

Result Value. The result is TSOURCE if MASK is true and FSOURCE otherwise.

Example. If TSOURCE is the array $\begin{bmatrix}1&6&5\\2&4&6\end{bmatrix}$, FSOURCE is the array $\begin{bmatrix}0&3&2\\7&4&8\end{bmatrix}$ and MASK is the array $\begin{bmatrix}T&.&T\\.&.&T\end{bmatrix}$, where "T" represents .TRUE. and "." represents .FALSE., then MERGE (TSOURCE, FSOURCE, MASK) is $\begin{bmatrix}1&3&5\\7&4&6\end{bmatrix}$.

13.12.62 MIN (A1, A2, A3, ...).

Optional Arguments. A3, ...

Description. Minimum value.

Kind. Elemental function.

Arguments. The arguments must all be of the same type which must be integer or real and they must all have the same type parameters.

Result Type and Type Parameters. Same as the arguments.

Result Value. The value of the result is that of the smallest argument.

Example. MIN (−9.0, 7.0, 2.0) has the value −9.0.

13.12.63 MINEXPONENT (X).

Description. Returns the minimum (most negative) exponent in the model representing numbers of the same type and type parameters as the argument.

Kind. Inquiry function.

Argument. X must be of type real. It may be scalar or array valued.

Result Type and Shape. Integer scalar.

Result Value. The result has value e_{min}, as defined in 13.6.1 for the model representing numbers of the same type and type parameters as X.

Example. MINEXPONENT (X) has the value −126 for real X whose model is as at the end of 13.6.1.

13.12.64 MINLOC (ARRAY, MASK).

Optional Argument. MASK

Description. Determine the location of an element of ARRAY having the minimum value of the elements identified by MASK.

Kind. Transformational function.

Arguments.

ARRAY must be of type integer or real. It must not be scalar.

MASK (optional) must be of type logical and must be conformable with ARRAY.

Result Type and Shape. The result is of type integer; it is an array of rank one and of size equal to the rank of ARRAY.

Result Value.

Case (i): If MASK is absent, the result is a rank-one array whose element values are the values of the subscripts of an element of ARRAY whose value equals the minimum value of all the elements of ARRAY. The *i*th subscript returned lies in the range 1 to e_i, where e_i is the extent of the *i*th dimension of ARRAY. If more than one element has minimum value, the element whose subscripts are returned is processor dependent. If ARRAY has size zero, the value of the result is processor dependent.

Case (ii): If MASK is present, the result is a rank-one array whose element values are the values of the subscripts of an element of ARRAY, corresponding to a true element of MASK, whose value equals the minimum value of all such elements of ARRAY. The *i*th subscript returned lies in the range 1 to e_i, where e_i is the extent of the *i*th dimension of ARRAY. If more than one such element has minimum value, the element whose subscripts are returned is processor dependent. If ARRAY has size zero or every element of MASK has the value .FALSE., the value of the result is processor dependent.

Examples.

Case (i): The value of MINLOC ([2, 4, 6]) is [1].

Case (ii): If A has the value $\begin{bmatrix} 0 & -5 & 8 & -3 \\ 3 & 4 & -1 & 2 \\ 1 & 5 & 6 & -4 \end{bmatrix}$, MINLOC (A, MASK = A.GT. −4) has the value [1,4].

13.12.65 MINVAL (ARRAY, DIM, MASK).

Optional Arguments. DIM, MASK

Description. Minimum value of all the elements of ARRAY along dimension DIM corresponding to true elements of MASK.

Kind. Transformational function.

Arguments.

ARRAY must be of type integer or real. It must not be scalar.

DIM (optional) must be scalar and of type integer with value in the range $1 \le \text{DIM} \le n$, where n is the rank of ARRAY.

MASK (optional) must be of type logical and must be conformable with ARRAY.

Result Type, Type Parameters, and Shape. The result is of the same type and type parameters as ARRAY. It is scalar if DIM is absent or ARRAY has rank one; otherwise, the result is an array of rank $n-1$ and of shape $(d_1, d_2, \ldots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \ldots, d_n)$ where $(d_1, d_2, \ldots, d_n)$ is the shape of ARRAY.

Result Value.

Case (i): The result of MINVAL (ARRAY) has value equal to the minimum value of all the elements of ARRAY or has value HUGE (ARRAY) if ARRAY has size zero.

Case (ii): The result of MINVAL (ARRAY, MASK = MASK) has value equal to the minimum value of the elements of ARRAY corresponding to true elements of

MASK or has value HUGE (ARRAY) if there are no true elements.

Case (iii): If ARRAY has rank one, MINVAL (ARRAY, DIM [,MASK]) has value equal to that of MINVAL (ARRAY [,MASK = MASK]). Otherwise, the value of element $(s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n)$ of MINVAL (ARRAY, DIM [,MASK]) 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].

13.12.66 MOD (A, P).

Description. Remainder modulo P.

Kind. Elemental function.

Arguments.

A	must be of type integer or real.
P	must be of the same type as A.

Result Type and Type Parameters. Same as A.

Result Value. If $P \neq 0$, the value of the result is A – INT (A/P) * P. If P = 0, the result is undefined.

Example. MOD (3.0, 2.0) has the value 1.0.

13.12.67 NEAREST (X, S).

Description. Returns the nearest different machine representable number in a given direction.

Kind. Elemental function.

Arguments.

X	must be of type real.
S	must be of type real and not equal to zero.

Result Type and Type Parameters. Same as X.

Result Value. The result has value equal to the machine representable number distinct from X and nearest to it in the direction of the infinity with the same sign as S.

Example. NEAREST (3.0, 2.0) has the value $3+2^{-22}$ on a machine whose representation is that of the model at the end of 13.6.1.

13.12.68 NINT (A).

Description. Nearest integer.

Kind. Elemental function.

Argument. A must be of type real.

Result Type. Integer.

Result Value. If A > 0, NINT (A) has the value INT (A+0.5); if A ≤ 0, NINT (A) has the value INT (A−0.5).

Example. NINT (2.783) has the value 3.

13.12.69 PACK (ARRAY, MASK, VECTOR).

Optional Argument. VECTOR

Description. Pack an array into an array of rank one under the control of a mask.

Kind. Transformational function.

Arguments.

ARRAY	may be of any type. It must not be scalar.
MASK	must be of type logical and must be conformable with ARRAY.
VECTOR (optional)	must be of the same type and type parameters as ARRAY and must have rank one. VECTOR must have at least as many elements as there are true elements in MASK. If MASK is scalar with value true, VECTOR must have at least as many elements as there are in ARRAY.

Result Type, Type Parameters, and Shape. The result is an array of rank one with the same type and type parameters as ARRAY. If VECTOR is present, the result size is that of VECTOR; otherwise, the result size is the number t of true elements in MASK unless MASK is scalar with value true, in which case the result size is the size of ARRAY.

Result Value. Element i of the result is the element of ARRAY that corresponds to the ith true element of MASK, taking elements in array element order, for $i = 1, 2, \ldots, t$. If VECTOR is present and has size $n > t$, element i of the result has value VECTOR (i), for $i = t + 1, \ldots, n$.

Example. The nonzero elements of an array M with value $\begin{bmatrix} 0 & 0 & 0 \\ 9 & 0 & 0 \\ 0 & 0 & 7 \end{bmatrix}$ may be "gathered" by the function PACK. The result of PACK (M, MASK=M.NE.0) is [9, 7] and the result of PACK (M, M.NE.0, VECTOR=[6[0]]) is [9, 7, 0, 0, 0, 0].

13.12.70 PRESENT (A).

Description. Determine whether an optional argument is present.

Kind. Inquiry function

Argument. A must be an optional argument of the procedure in which the PRESENT function reference appears.

Result Type and Shape. Logical scalar.

Result Value. The result has the value .TRUE. if A is present (12.5.2.8) and is otherwise .FALSE.

13.12.71 PRODUCT (ARRAY, DIM, MASK).

Optional Arguments. DIM, MASK

Description. Product of all the elements of ARRAY along dimension DIM corresponding to the true elements of MASK.

Kind. Transformational function.

Arguments.

ARRAY must be of type integer, real, or complex. It must not be scalar. Its shape must be defined.

DIM (optional) must be scalar and of type integer with value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY.

MASK (optional) must be of type logical and must be conformable with ARRAY.

Result Type, Type Parameters, and Shape. The result is of the same type and type parameters as ARRAY. It is scalar if DIM is absent or ARRAY has rank one; otherwise, the result is an array of rank $n-1$ and of shape $(d_1, d_2, \ldots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \ldots, d_n)$ where $(d_1, d_2, \ldots, d_n)$ is the shape of ARRAY.

Result Value.

Case (i): The result of PRODUCT (ARRAY) has value equal to a processor-dependent approximation to the product of all the elements of ARRAY or has value one if ARRAY has size zero.

Case (ii): The result of PRODUCT (ARRAY, MASK = MASK) has value equal to a processor-dependent approximation to the product of the elements of ARRAY corresponding to true elements of MASK or has value one if there are no true elements.

Case (iii): If ARRAY has rank one, PRODUCT (ARRAY, DIM [,MASK]) has value equal to that of PRODUCT (ARRAY [,MASK = MASK]). Otherwise, the value of element $(s_1, s_2, \ldots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \ldots, s_n)$ of PRODUCT (ARRAY, DIM [,MASK]) is equal to PRODUCT (ARRAY $(s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)$ [, MASK = MASK $(s_1, s_2, \ldots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \ldots, s_n)$]).

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].

13.12.72 RADIX (X).

Description. Returns the base of the model representing numbers of the same type and type parameters as the argument.

Kind. Inquiry function.

Argument. X must be of type integer or real. It may be scalar or array valued.

Result Type and Shape. Integer scalar.

Result Value. The result has value r if X is of type integer and b if X is of type real, where r and b are as defined in 13.6.1 for the model representing numbers of the same type and type parameters as X.

Example. RADIX (X) has the value 2 for real X whose model is as at the end of 13.6.1.

13.12.73 RANDOM (HARVEST).

Description. Returns one pseudorandom number or an array of pseudorandom numbers from the uniform distribution over the range $0 \leq x < 1$.

Kind. Subroutine.

Argument. HARVEST must be of type real. It may be a scalar or an array variable. It is set to contain pseudorandom numbers from the uniform distribution in the interval

$0 \leq x < 1$.

Examples.

```
REAL X, Y (10, 10)
CALL RANDOM (HARVEST = X)  ! INITIALIZES X WITH A PSEUDORANDOM NUMBER
CALL RANDOM (Y)
! X AND Y CONTAIN UNIFORMLY DISTRIBUTED RANDOM NUMBERS
```

13.12.74 RANDOMSEED (SIZE, PUT, GET).

Optional Argument. SIZE, PUT, GET

Description. Initializes or restarts the pseudorandom number generator.

Kind. Subroutine.

Arguments. There must either be exactly one or no arguments present.

SIZE (optional) must be scalar and of type integer. It is set to the number *N* of integers that the processor uses to hold the value of the seed.

PUT (optional) must be an integer array of rank one and size $\geq N$. It is used by the processor to set the seed value.

GET (optional) must be an integer array of rank one and size $\geq N$. It is set by the processor to the current value of the seed.

If no argument is present, the processor sets the seed to a processor-determined value.

Examples.

```
CALL RANDOMSEED  ! PROCESSOR INITIALIZATION
CALL RANDOMSEED (SIZE = K)  ! SETS K = N
CALL RANDOMSEED (PUT = SEED (1 : K))  ! SET USER SEED
CALL RANDOMSEED (GET = OLD (1 : K))  ! READ CURRENT SEED
```

13.12.75 REAL (A, MOLD).

Optional Argument. MOLD

Description. Convert to real type.

Kind. Elemental function.

Arguments.

A must be of type integer, real, or complex.

MOLD (optional) must be of type real or complex.

Result Type and Type Parameters. Real. If MOLD is present, the type parameters are those of MOLD; otherwise, they are the processor-dependent type parameters for the default real type.

Result Value.

Case (i): If A is of type integer or real, the result is equal to a processor-dependent approximation to A.

Case (ii): If A is of type complex, the result is equal to a processor-dependent approximation to the real part of A.

Examples. REAL (−3) has the value −3.0. REAL (Z, Z) has the same type parameters and the same value as the real part of the complex variable Z.

13.12.76 REPEAT (STRING, NCOPIES).

Description. Concatenate several copies of a string.

Kind. Elemental function.

Arguments.

STRING	must be of type character.
NCOPIES	must be of type integer. Its value must not be negative.

Result Type and Type Parameters. Character of length NCOPIES times that of STRING.

Result Value. The value of the result is the concatenation of NCOPIES copies of STRING.

Example. REPEAT ('H', 2) has value 'HH'.

13.12.77 RESHAPE (MOLD, SOURCE, PAD, ORDER).

Optional Arguments. PAD, ORDER

Description. Change the shape of an array.

Kind. Transformational function.

Arguments.

MOLD	must be of type integer and rank one. Its size must be positive and less than 8. It must not have an element whose value is negative.
SOURCE	may be of any type. It must be array valued. If PAD is absent or of size zero, the size of SOURCE must be ≥ PRODUCT (MOLD). The size of the result is the product of the values of the elements of MOLD.
PAD (optional)	must be of the same type and type parameters as SOURCE. PAD must be array valued.
ORDER (optional)	must be of type integer, must have the same shape as MOLD, and its value must be a permutation of [1:*n*], where *n* is the size of MOLD. If absent, it is as if it were present with value [1:*n*].

Result Type, Type Parameters, and Shape. The result is an array of shape MOLD (i.e., DSHAPE (RESHAPE (MOLD, SOURCE, PAD, ORDER)) is equal to MOLD) with type and type parameters those of SOURCE.

Result Value. The elements of the result, taken in permuted subscript order ORDER (1), ..., ORDER (*n*), are those of SOURCE in normal 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.

Example. RESHAPE ([2, 3], [1:6]) has value $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$.

13.12.78 RRSPACING (X).

Description. Returns the reciprocal of the relative spacing of model numbers near the argument value.

Kind. Elemental function.

Argument. X must be of type real.

Result Type and Type Parameters. Same as X.

Result Value. The result has value $|X \times b^{-e}| \times b^{p}$, where b, e, and p are as defined in 13.6.1 for the model representation of X, provided this result is within range.

Example. RRSPACING (−3.0) has the value 0.75×2^{24} for reals whose model is as at the end of 13.6.1.

13.12.79 SCALE (X, I).

Description. Returns $X \times b^{I}$ where b is the base in the model representation of X.

Kind. Elemental function.

Arguments.

X	must be of type real.
I	must be of type integer.

Result Type and Type Parameters. Same as X.

Result Value. The result has the value $X \times b^{I}$, where b is defined in 13.6.1 for model numbers representing values of X, provided this result is within range.

Example. SCALE (3.0, 2) has the value 12.0 for reals whose model is as at the end of 13.6.1.

13.12.80 SCAN (STRING, SET, BACK).

Optional Argument. BACK

Description. Scan a string for a character in a set of characters.

Kind. Elemental function.

Arguments.

STRING	must be of type character.
SET	must be of type character.
BACK (optional)	must be of type logical.

Result Type. Integer.

Result Value.

Case (i): If any of the characters of SET appears in STRING, the value of the result is the position of the leftmost character of STRING that is in SET. The result is zero if STRING does not contain any of the characters that are in SET or if the length of STRING or SET is zero. The default value of BACK is .FALSE. and its inclusion is optional when processing starts with the first character of STRING.

Case (ii): If STRING is to be processed starting with the last character, BACK must contain the logical value .TRUE. The value of the result is the position of the rightmost character of STRING that is in SET. The result is zero if STRING does not contain any of the characters that are in SET or if the length of STRING or SET is zero.

Examples. SCAN ('FORTRAN', 'TR') has value 3. SCAN ('FORTRAN', 'TR', BACK = .TRUE.) has the value 5.

13.12.81 SETEXPONENT (X, I).

Description. Returns the model number whose fractional part is the fractional part of the model representation of X and whose exponent part is I.

Kind. Elemental function.

Arguments.

X	must be of type real.
I	must be of type integer.

Result Type and Type Parameters. Same as X.

Result Value. The result has value $X \times b^{I-e}$, where b and e are as defined in 13.6.1 for the model representation of X, provided this result is within range. If X has value zero, the result has value zero.

Example. SETEXPONENT (3.0, 1) has the value 1.5 for reals whose model is as at the end of 13.6.1.

13.12.82 SIGN (A, B).

Description. Absolute value of A times the sign of B.

Kind. Elemental function.

Arguments.

A	must be of type integer or real.
B	must be of the same type as A.

Result Type and Type Parameters. Same as A.

Result Value. The value of the result is $|A|$ if $B \geq 0$ and $-|A|$ if $B < 0$.

Example. SIGN (−3.0, 2.0) has the value 3.0.

13.12.83 SIN (X).

Description. Sine function.

Kind. Elemental function.

Argument. X must be of type real or complex.

Result Type and Type Parameters. Same as X.

Result Value. The result has value equal to a processor-dependent approximation to sin(X). If X is of type real, 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).

13.12.84 SINH (X).

Description. Hyperbolic sine function.

Kind. Elemental function.

Argument. X must be of type real.

Result Type and Type Parameters. Same as X.

Result Value. The result has value equal to a processor-dependent approximation to sinh(X).

Example. SINH (1.0) has the value 1.1752012 (approximately).

13.12.85 SPACING (X).

Description. Returns the absolute spacing of model numbers near the argument value.

Kind. Elemental function.

Argument. X must be of type real.

Result Type and Type Parameters. Same as X.

Result Value. The result has value b^{e-p}, where b, e, and p are as defined in 13.6.1 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 13.6.1.

13.12.86 SPREAD (SOURCE, DIM, NCOPIES).

Description. Replicates an array by adding a dimension. Broadcasts several copies of SOURCE along a specified dimension (as in forming a book from copies of a single page) and thus forms an array of rank one greater.

Kind. Transformational function.

Arguments.

SOURCE	may be of any type. It may be scalar or array valued. The rank of SOURCE must be less than 7.
DIM	must be scalar and of type integer with value in the range $1 \le \text{DIM} \le n+1$, where n is the rank of SOURCE.
NCOPIES	must be scalar and of type integer.

Result Type, Type Parameters, and Shape. The result is an array of the same type and type parameters as SOURCE and of rank $n+1$, where n is the rank of SOURCE.

Case (i): If SOURCE is scalar, the shape of the result is [MAX (NCOPIES, 0)].

Case (ii): If SOURCE is array valued with shape E (1:n), the shape of the result is [E (1:DIM-1), MAX (NCOPIES, 0), E (DIM:n)].

Result Value.

Case (i): If SOURCE is scalar, each element of the result has value equal to SOURCE.

Case (ii): If SOURCE is array valued, the element of the result with subscripts (r_1, r_2, ..., r_{n+1}) has the value SOURCE (s_1, s_2, ..., s_n) where (s_1, s_2, ..., s_n) is (r_1, r_2, ..., r_{n+1}) with subscript r_{DIM} omitted.

Example. If A is the array [2, 3, 4], SPREAD (A, DIM = 1, NCOPIES = 3) is the array $\begin{bmatrix} 2 & 3 & 4 \\ 2 & 3 & 4 \\ 2 & 3 & 4 \end{bmatrix}$.

13.12.87 SQRT (X).

Description. Square root.

Kind. Elemental function.

Argument. X must be of type real or complex. Unless X is complex, its value must be greater than or equal to zero.

Result Type and Type Parameters. Same as X.

Result Value. The result has value equal to a processor-dependent approximation to the square root of X. A result of type complex is the principal value with the real part greater than or equal to zero. When the real part of the result is zero, the imaginary part is greater than or equal to zero.

Example. SQRT (4.0) has the value 2.0 (approximately).

13.12.88 SUM (ARRAY, DIM, MASK).

Optional Arguments. DIM, MASK

Description. Sum all the elements of ARRAY along dimension DIM with mask MASK.

Kind. Transformational function.

Arguments.

ARRAY	must be of type integer, real, or complex. It must not be scalar.
DIM (optional)	must be scalar and of type integer with value in the range $1 \le \text{DIM} \le n$, where n is the rank of ARRAY.
MASK (optional)	must be of type logical and must be conformable with ARRAY.

Result Type, Type Parameters, and Shape. The result is of the same type and type parameters as ARRAY. It is scalar if DIM is absent or ARRAY has rank one; otherwise, the result is an array of rank $n-1$ and of shape $(d_1, d_2, ..., d_{\text{DIM}-1}, d_{\text{DIM}+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of ARRAY.

Result Value.

Case (i): The result of SUM (ARRAY) has value equal to a processor-dependent approximation to the sum of all the elements of ARRAY or has value zero if ARRAY has size zero.

Case (ii): The result of SUM (ARRAY, MASK = MASK) has value equal to a processor-dependent approximation to the sum of the elements of ARRAY corresponding to the true elements of MASK or has value zero if there are no true elements.

Case (iii): If ARRAY has rank one, SUM (ARRAY, DIM [,MASK]) has value equal to that of SUM (ARRAY [,MASK = MASK]). Otherwise, the value of element $(s_1, s_2, ..., s_{\text{DIM}-1}, s_{\text{DIM}+1}, ..., s_n)$ of SUM (ARRAY, DIM [,MASK]) is equal to SUM (ARRAY $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$ [, MASK= MASK $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$]).

Examples.

Case (i): The value of SUM ([1, 2, 3]) is 6.

Case (ii): SUM (C, MASK= C .GT. 0.0) forms the arithmetic sum of the positive elements of C.

Case (iii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$, SUM (B, DIM=1) is [3, 7, 11] and SUM (B, DIM=2) is [9, 12].

13.12.89 SYSTEM_CLOCK (COUNT, COUNT_RATE, COUNT_MAX).

Optional Arguments. COUNT, COUNT_RATE, COUNT_MAX

Description. Returns integer data from a real-time clock.

Kind. Subroutine.

Arguments.

COUNT (optional)	must be scalar and of type integer. It is set to a processor-dependent value based on the current value of the basic clock or to –HUGE (0) if there is no clock. The processor-dependent value is incremented by one for each clock count until the value COUNT_MAX is reached and is reset to zero at the next count. It lies in the range 0 to COUNT_MAX if there is a clock.

COUNT__RATE (optional)
must be scalar and of type integer. It is set to the number of basic clock counts per second, or to zero if there is no clock.

COUNT__MAX (optional)
must be scalar and of type integer. It is set to the maximum value that COUNT can have, or to zero if there is no clock.

Example. If the basic system clock is a 24-hour clock that registers time in 1-second intervals, at 11:30 A.M. the reference

```
CALL SYSTEM_CLOCK (COUNT = C, COUNT_RATE = R, COUNT_MAX = M)
```

sets C = 11 × 3600 + 30 × 60 = 41400, R = 1, and M = 24 × 3600 − 1 = 86399.

13.12.90 TAN (X).

Description. Tangent function.

Kind. Elemental function.

Argument. X must be of type real.

Result Type and Type Parameters. Same as X.

Result Value. The result has value equal to a processor-dependent approximation to tan(X), with X regarded as a value in radians.

Example. TAN (1.0) has the value 1.5574077 (approximately).

13.12.91 TANH (X).

Description. Hyperbolic tangent function.

Kind. Elemental function.

Argument. X must be of type real.

Result Type and Type Parameters. Same as X.

Result Value. The result has value equal to a processor-dependent approximation to tanh(X).

Example. TANH (1.0) has the value 0.76159416 (approximately).

13.12.92 TINY (X).

Description. Returns the smallest positive number in the model representing numbers of the same type and type parameters as the argument.

Kind. Inquiry function.

Argument. X must be of type real. It may be scalar or array valued.

Result Type, Type Parameters, and Shape. Scalar with the same type and type parameters as X.

Result Value. The result has value $b^{e_{min}-1}$ where b and e_{min} are as defined in 13.6.1 for the model representing numbers of the same type and type parameters as X.

Example. TINY (X) has the value 2^{-127} for real X whose model is as at the end of 13.6.1.

13.12.93 TRANSFER (SOURCE, MOLD, SIZE).

Optional Argument. SIZE

Description. Returns a result with a physical representation identical to that of SOURCE but interpreted with the type and type parameters of MOLD.

Kind. 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)	must be scalar and of type integer.

Result Type, Type Parameters, and Shape. 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) must be that of E. IF D is an array and E is an array of rank one, the value of TRANSFER (TRANSFER (E, D), E, ESIZE (E)) must be that 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 value (1.1, 2.2) and whose second element has real part with value 3.3.

Case (iii): TRANSFER ([1.1, 2.2, 3.3], [(0.0, 0.0)], 1) has value [(1.1, 2.2)].

13.12.94 TRANSPOSE (MATRIX).

Description. Transpose an array of rank two.

Kind. Transformational function.

Argument. MATRIX may be of any type and must have rank two. Its shape must be defined.

Result Type, Type Parameters, and Shape. The result is an array of the same type and type parameters as MATRIX and with rank two and shape [*n*, *m*] where [*m*, *n*] is the shape of MATRIX.

Result Value. Element (*i*, *j*) of the result has value MATRIX (*j*, *i*), $i = 1, 2, \ldots, n$; $j = 1, 2, \ldots, m$.

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 $\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$.

13.12.95 TRIM (STRING).

Description. Returns the argument with trailing blank characters removed.

Kind. Transformational function.

Argument. STRING must be of type character and must be a scalar.

Result Type and Type Parameters. Character with a length that is the length of STRING less the number of trailing blanks in STRING.

Result Value. The value of the result is the same as STRING except any trailing blanks are removed. If STRING contains no nonblank characters, the result has zero length.

Example. TRIM (' A B ') has value ' A B'.

13.12.96 UNPACK (VECTOR, MASK, FIELD).

Description. Unpack an array of rank one into an array under the control of a mask.

Kind. Transformational function.

Arguments.

VECTOR	may be of any type. It must have rank one. Its size must be at least *t* where *t* is the number of true elements in MASK.
MASK	must be array valued and of type logical. Its shape must be defined.
FIELD	must be of the same type and type parameters as VECTOR and must be conformable with MASK.

Result Type, Type Parameters, and Shape. The result is an array of the same type and type parameters as VECTOR and the same shape as MASK.

Result Value. The element of the result that corresponds to the *i*th true element of MASK, in array element order, has value VECTOR (*i*) for $i = 1, 2, \ldots, t$, where t is the number of true values in MASK. Other elements have value equal to FIELD if FIELD is scalar or to the corresponding element of FIELD if it is an array.

Example. Specific values may be "scattered" to specific positions in an array by using UNPACK. If M is the array $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$, V is the array [1, 2, 3], and Q is the logical mask $\begin{bmatrix} . & T & . \\ T & . & . \\ . & . & T \end{bmatrix}$, where "T" represents .TRUE. and "." represents .FALSE., then the result of UNPACK (V, MASK = Q, FIELD = M) has the value $\begin{bmatrix} 0 & 2 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 3 \end{bmatrix}$ and the result of UNPACK (V, MASK = Q, FIELD = 0) has the value $\begin{bmatrix} 0 & 2 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 3 \end{bmatrix}$.

13.12.97 VERIFY (STRING, SET, BACK).

Optional Argument. BACK

Description. Verify that a set of characters contains all the characters in a string.

Kind. Elemental function.

Arguments.

STRING	must be of type character.

SET must be of type character.

BACK (optional) must be of type logical.

Result Type. Integer.

Result Value. The value of the result is zero if each character in STRING is in SET or if STRING has zero length; otherwise, there are two cases as follows:

Case (i): If BACK is absent or present with the value .FALSE., 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., the value of the result is the position of the rightmost character of STRING that is not in SET.

Examples. VERIFY ('ABBA', 'AB') has value 0.

Case (i): VERIFY ('ABBA', 'A') has the value 2.

Case (ii): VERIFY ('ABBA', 'A', BACK = .TRUE.) has the value 3.

14. SCOPE, ASSOCIATION, AND DEFINITION

Each lexical token has a **scope**, which is either an executable program, a scoping unit, a single statement, or part of a statement. Within its scope, a lexical token has a single interpretation. An entity identified by a lexical token whose scope is an executable program is called a **global entity**. An entity identified by a lexical token whose scope is a scoping unit (2.2.1) is called a **local entity**. An entity identified by a lexical token whose scope is a single statement or part of a statement is called a **statement entity**.

By means of association, a named entity may be known by the same name or a different name in a different scoping unit, or by a different name in the same scoping unit.

14.1. Scope of Names. The names of external procedures, common blocks, and program units have a scope of an executable program.

The names of variables, constants, statement functions, internal procedures, module procedures, dummy procedures, intrinsic procedures, keyword arguments, derived types, type parameters, type components, range lists, namelist groups, and constructs have a scope of a scoping unit.

The name of a variable that appears as an IDENTIFY subscript or as a dummy argument in a statement function statement has a scope of the statement in which it appears.

The name of a variable that appears as the DO variable of an implied-DO in a DATA statement or INQUIRE statement has a scope of the implied-DO list.

14.1.1. Global Entities. Program units, common blocks, and external procedures are global entities of an executable program. A name that identifies a global entity must not be used to identify any other global entity in the same executable program.

14.1.2. Local Entities. Within a scoping unit, entities in the following classes:

(1) Named variables, named constants, constructs, statement functions, internal procedures, module procedures, dummy procedures, intrinsic procedures, derived types, range lists, and namelist group names,

(2) Type parameters, in a separate class for each type,

(3) Type components, in a separate class for each type, and

(4) Argument keywords, in a separate class for each procedure with an explicit interface

are local entities of that scoping unit.

A name that identifies a global entity in a scoping unit must not be used to identify a local entity of class (1) in that scoping unit, except for a common block name (14.1.2.1) or an external function name (14.1.2.2).

Within a scoping unit, a name that identifies a local entity of one class must not be used to identify another entity of the same class, except in the case of overloaded procedures (14.1.2.3). A name that identifies a local entity of one class may be used to identify a local entity of another class.

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.

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 function, or a local variable that is also an external function in a function subprogram. If a name is used for both a common block and a local entity, the appearance of that name in any context other than as a common block name in a COMMON or SAVE statement identifies only the local entity. Note that an intrinsic function name may be a common block name in a scoping unit that does not

reference the intrinsic function.

14.1.2.2. Function Results. If a function subprogram does not have a RESULT clause in its function statement, there must be a local variable with the same name as that function. If a function subprogram contains an ENTRY statement, there must be a local variable with the same name as the entry.

14.1.2.3. Procedure Overloading. Within a scoping unit, two procedures may have the same name provided they both have explicit interfaces and at least one of them has a nonoptional dummy argument which

(1) Corresponds by position in the argument list to a dummy argument not present in the other, present with a different type, present with different type parameters, or present with a different rank when both are allocatable or assumed-shape arrays; and

(2) Corresponds by argument keyword to a dummy argument not present in the other, present with a different type, present with different type parameters, or present with a different rank when both are allocatable or assumed-shape arrays.

For example, the procedures with interfaces

```
INTERFACE
   SUBROUTINE A (X)
   REAL X
END INTERFACE

INTERFACE
   SUBROUTINE A (I)
   INTEGER I
END INTERFACE
```

satisfy rules (1) and (2) and therefore the procedures may be overloaded. However, if I were declared REAL, rule (1) would not be satisfied while rule (2) remains satisfied; this case is not allowed because the reference to A in the statement

```
CALL A (0.0)
```

would be ambiguous.

14.1.2.4. Components. A component name has the same scope as the type of which it is a component. 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 or host association (14.7.1.2) 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.

14.1.2.5. Type Parameters. A type parameter name has the same scope as the type of which it is a parameter. This name is either the name of a derived-type intrinsic inquiry function (13.4.5) or is the name of a derived-type keyword argument (5.1.1.7). There is also a variable of the same name whose scope is the derived-type definition. As a type parameter name, it may appear only in a derived-type declaration, a derived-type constructor for the type of which it is a parameter, and as the name of the inquiry function (13.4.5) for the parameter value for a structure of that type. If the type is accessible in another scoping unit by use or host association (14.7.1.2), the type parameter name is accessible for derived-type declarations, derived-type constructors, and inquiries for that type in that scoping unit. The inquiry function associated with the type parameter name has the same scope as the type. Note that in the case of a constructor, the components of the type must also be accessible.

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 its host. 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 or host association (14.7.1.2), the argument keyword is accessible for procedure references for that procedure in that scoping unit.

14.1.3. Statement Entities. 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 that it would have if it were the name of a variable in the scoping unit that includes the statement function.

The name of an IDENTIFY subscript has a scope of that IDENTIFY statement. It is always of type integer.

The name of a variable that appears as the DO variable of an implied-DO in a DATA statement has a scope of the implied-DO list. It has the type that it would have if it were the name of a variable in the scoping unit that includes the DATA statement and this type must be integer.

The name of a statement entity also may be the name of a global or local entity in the same scoping unit; in this case, the name is interpreted within its statement scope as that of the statement variable.

14.2. Scope of Labels. A label has a scope of a scoping unit. No two statements in the same scoping unit may have the same label.

14.3. Scope of Exponent Letters. An exponent letter has a scope of a scoping unit. It also may be the name of a global or local entity in the same scoping unit.

14.4. Scope of External Input/Output Units. An external input/output unit has the scope of an executable program.

14.5. Scope of Operators. The intrinsic operators have a scope of an executable program. A defined operator has a scope of a scoping unit. Within a scoping unit, two operations may be identified by the same operator provided they have at least one corresponding operand with different type, different type parameters, or different rank.

14.6. Scope of the Assignment Symbol. Intrinsic assignment has a scope of an executable program. A defined assignment has a scope of a scoping unit. Within a scoping unit, two assignments may be identified by the assignment symbol provided they have at least one corresponding operand with different type, different type parameters, or different rank.

14.7. Association. Two entities may become associated by name association or by storage association. When entities become associated, each part of one is associated with the corresponding part of the other.

14.7.1. Name Association. There are four forms of **name association**: argument association, alias association, use association, and host association. Argument, use, and host association provide mechanisms by which entities known in a scoping unit may be accessed in another scoping unit. Alias association provides alternative means (for example, different names) of access to a data object within a single scoping unit.

14.7.1.1 Argument Association. The rules governing argument association are given in Section 12. As explained in Section 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. (Note that an actual argument may be a nameless data entity, such as an expression that is not simply a variable or constant.) The dummy argument name is the name by which the associated actual argument is known, and by which it may be accessed, in the called procedure.

Upon termination of execution of a procedure reference, all argument associations established by that reference are terminated. A dummy argument of that procedure may be associated with an entirely different actual argument in a subsequent execution of the procedure.

14.7.1.2 Use and Host 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.

The rules for host association are given in 11.2.2.

Use or host association allows access in one scoping unit to entities defined in another scoping unit and remains in effect throughout the execution of the executable program.

Within a scoping unit, an entity accessed by use or host association must not appear in a type declaration statement or otherwise have any of its attributes specified. It assumes all attributes, and only those attributes, of its associated entity. If the entity is renamed in a USE statement in a scoping unit, the original name is not associated with it in this scoping unit and may be used for other purposes. The new local name may be used in exactly the same way as the original name could have been used if there had been no renaming.

14.7.1.3 Alias Association. Alias association provides another form of name association, in addition to argument, use, and host association.

An alias provides an alternative access to a data object within a single scoping unit. The process of establishing an alias and the resulting relationship is known as **alias association**.

The rules for alias association are given in 6.2.6. The alias name must have the ALIAS attribute.

An alias association between an alias and a nonalias object is established upon execution of an IDENTIFY statement and continues thereafter until the first occurrence of:

(1) Execution of another IDENTIFY statement in the same scoping unit involving the same alias as an alias name,

(2) Termination of execution of the scoping unit, or

(3) Deallocation of the associated nonalias data object.

An alias may be associated with any nonalias data object that has the same type and type parameters. The association may be established through an existing alias. An alias must not be referenced or defined unless it is alias associated. An alias association with an allocatable array must not be established unless the allocatable array is allocated. Deallocation of an allocatable array terminates all alias associations with it.

Any number of aliases may be associated concurrently with a given nonalias object. Each such alias provides access to the associated data object, and the nonalias object continues to be accessible by its original name. An alias may be reassociated by IDENTIFY statements any number of times with the same data object during execution of a scoping unit.

Summary Comparison of Alias, Use, and Host Associations

Characteristic	Alias Associations	Use Associations	Host Associations
Scope	Single scoping unit	Single scoping unit, plus using scoping units if in a module	Single scoping unit plus scoping units of internal or module subprograms
Duration	Temporary	Entire program execution	Entire program execution
May change?	Yes	No	No
How established?	Execution of IDENTIFY statement	Appearance in USE statement	Internal or module subprogram or interface block
How terminated?	Execution of IDENTIFY statement Deallocation of the entity Termination of execution of the executable program	Termination of execution of the executable program	Termination of execution of the executable program
Appearance in USE tatement	Not allowed	Normal (only) way to establish	Not allowed
Appearance in IDENTIFY statement	As alias variable As parent variable	As parent variable	As alias variable As parent variable
Allowed with unallocated parent	No	Yes	Yes
May appear in ALLOCATE statement	No, but parent may	Yes	Yes
May appear in DEALLOCATE statement	No, but parent may	Yes	Yes

14.7.2. Storage Association. Storage sequences are used to describe relationships that exist among variables, array elements, substrings, common blocks, and arguments. **Storage association** is the association of two or more data objects that occurs when two or more storage sequences share one or more storage units.

14.7.2.1. Storage Sequence. A **storage sequence** is a sequence of storage units. The **size of a storage sequence** is the number of storage units in the storage sequence. A **storage unit** is a character storage unit or a numeric storage unit.

A scalar object of type integer, default real, or logical has a storage sequence of one numeric storage unit.

A variable or array element of a derived type has no storage sequence.

A structure component, or an element of it if an array, has no storage sequence, even if of an intrinsic type.

A real or complex object with explicitly specified precision and range attributes has no storage sequence.

A scalar object of type double precision real or default complex has a storage sequence of two numeric storage units. In a complex storage sequence, the real part has the first storage unit and the imaginary part has the second storage unit.

A scalar object of type character has a storage sequence of character storage units. The number of character storage units in the storage sequence is the length of the character entity. The order of the sequence corresponds to the ordering of character positions (4.3.2.1 and 5.1.1.5).

Each common block has a storage sequence (5.5.2.1).

Each data object appearing in a storage association context has a storage sequence (2.4.8).

14.7.2.2. Association of Storage Sequences. Two storage sequences s_1 and s_2 are **storage associated** if the *i*th storage unit of s_1 is the same as the *j*th storage unit of s_2. This causes the $(i + k)$th storage unit of s_1 to be the same as the $(j + k)$th storage unit of s_2, for each integer k such that $1 \le i + k \le$ *size of* s_1 and $1 \le j + k \le$ *size of* s_2.

14.7.2.3. Association of Scalar Data Objects. Two scalar data objects are storage associated if their storage sequences are associated. Two scalar entities are **totally associated** if they have the same storage sequence. Two scalar entities are **partially associated** if they are associated but not totally associated.

The definition status and value of a data object affects the definition status and value of any associated entity. An EQUIVALENCE statement, a COMMON statement, an ENTRY statement, or a procedure reference may cause association of storage sequences.

An EQUIVALENCE statement causes association of data objects only within one 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 associated with data objects in another scoping unit.

In a function subprogram, an ENTRY statement causes the entry name to become associated with the name of the function subprogram which appears in the FUNCTION statement.

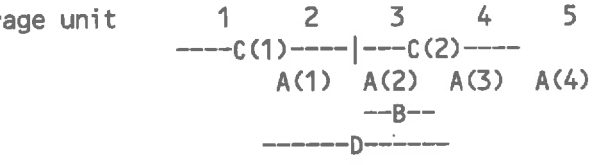
Partial association may exist only between two character entities or between a complex or double precision real entity and an entity of type integer, real, logical, double precision real, or complex.

Except for character entities, partial association may occur only through the use of COMMON, EQUIVALENCE, or ENTRY statements. Partial association must not occur through argument association, except for arguments of type character.

In the example:

```
REAL A (4), B
COMPLEX C (2)
DOUBLE PRECISION D
EQUIVALENCE (C (2), A (2), B), (A, D)
```

the third storage unit of C, the second storage unit of A, the storage unit of B, and the second storage unit of D are specified as the same. The storage sequences may be illustrated as:



A(2) and B are totally associated. The following are partially associated: A(1) and C(1), A(2) and C(2), A(3) and C(2), B and C(2), A(1) and D, A(2) and D, B and D, C(1) and D, and C(2) and D. Note that although C(1) and C(2) are each associated with D, C(1) and C(2) are not associated with each other.

Partial association of character entities occurs when some, but not all, of the storage units of the entities are the same. In the example:

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

A, B, and C are partially associated.

14.8. Definition and Undefinition of Variables. A variable may be defined or may be undefined and its definition status may change during execution of an executable 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.

14.8.1. Definition of Objects and Subobjects. Arrays, including sections, and variables of derived, character, complex, or default double precision real 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.

Execution of an IDENTIFY statement changes the association between the alias object and the parent object and therefore may change the definition status of the alias variable.

Execution of a SET RANGE statement changes the effective bounds of one or more arrays and therefore may change their definition status. Note in particular that an array variable as determined by its effective bounds might be defined, but a subobject of the array that consists of one or more elements from outside of the effective bounds might be undefined.

14.8.2. Variables That Are Always Defined. Zero-sized arrays and zero-length strings are always defined.

14.8.3. Variables That Are Initially Defined. The following variables are defined initially:

(1) Variables specified to have initial values by DATA statements,

(2) Variables specified to have initial values by type declaration statements with the DATA attribute, and

(3) Variables that are always defined.

14.8.4. Variables That Are Initially Undefined. All other variables are initially undefined.

14.8.5. Events That Cause Variables to Become Defined. Variables become defined as follows:

(1) Execution of an assignment statement other than a masked array assignment statement causes the variable that precedes the equals to become defined.

(2) Execution of a masked array assignment statement may cause some or all of the array elements in the assignment statement to become defined (7.5.2.2).

(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 (5) in 14.8.6.)

(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) Execution of an ASSIGN statement causes the variable in the statement to become defined with a statement label value.

(7) A reference to a procedure causes a subobject of a dummy argument to become defined if the corresponding subobject of the actual argument is defined with a value that is not a statement label.

(8) Execution of an input/output statement containing an input/output IOSTAT= specifier causes the specified integer variable to become defined.

(9) Execution of an input/output statement containing a NULLS= or VALUES= specifier causes the specified integer variable to become defined.

(10) Execution of an INQUIRE statement causes any variable that is assigned a value during the execution of the statement to become defined if no error condition exists.

(11) 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, except that variables associated with the variable in an ASSIGN statement become undefined when the ASSIGN statement is executed.

When a scalar variable without a storage sequence becomes defined, all associated variables become defined.

(12) Execution of an ALLOCATE or DEALLOCATE statement with a STAT= specifier causes the specified variable to become defined.

(13) Allocation of a zero-sized array causes the array to become defined.

(14) Invocation of a procedure causes any automatic object of zero size to become defined.

14.8.6 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) 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.

(3) 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.

(4) 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 accessed in a scoping unit that is currently in execution.

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

(5) 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, except those counted by a VALUES= specifier.

(6) When an error or end-of-file condition occurs during execution of an input/output statement, some or all of the implied DO variables may become undefined (9.4.3).

(7) Execution of a defined assignment statement may leave all or part of the variable that precedes the equal sign undefined.

(8) 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.

(9) Execution of an INQUIRE statement may cause the NAME=, RECL=, and NEXTREC= variables to become undefined (9.6).

(10) 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 a scalar variable without a storage sequence becomes undefined, all associated variables become undefined.

(11) A reference to a procedure causes a 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.

(12) When an allocatable array is deallocated, it becomes undefined. Successful execution of an ALLOCATE statement causes the elements of the array to become undefined.

(13) 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.

(14) 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.

(c) An actual argument associated with a dummy argument with intent OUT becomes undefined.

(d) A subobject of a dummy argument is undefined if the corresponding subobject of the actual argument is undefined.

APPENDIX A FORTRAN FAMILY OF STANDARDS

(This appendix is not part of American National Standard X3.9-198x, but is included for information only.)

A host language standard, such as Fortran, should take responsibility for coordinating other standards built on its base to prevent the development of conflicting collateral standards. A Fortran Reference Model has been suggested for the **Fortran Family of Standards**.

The Fortran Family of Standards consists of:

(1) The Fortran Language Standard

(2) Supplementary Standards based on Procedure Libraries

(3) Supplementary Standards based on Module Libraries

(4) Secondary Standards

X3.9-1978 (the previous Fortran standard) is referred to as **FORTRAN 77** in this appendix. This standard is referred to as **Fortran 8x**. A possible successor is referred to as **Fortran 9x.**

A.1 The Fortran Language Standard. The Fortran Language consists of **primary features** from FORTRAN 77, **decremental features** that are deleted, obsolescent, or deprecated in this standard, and **incremental features** that add new constructs to Fortran. (See Figure A.1.)

A.1.1 Primary Features. These features are those from the FORTRAN 77 standard that continue to be useful and characteristic of the language. Primary features are expected to continue throughout the life of Fortran or at least for the next several revisions of the language.

A.1.2 Incremental Features. These features are new to the language and are needed to improve the usefulness of Fortran. They are developed from current practice in extended Fortran implementations and in other contemporary languages.

The criteria for incremental features are:

(1) The feature is responsive to new system architectures.

(2) The feature improves the functionality of Fortran.

(3) The feature is desirable for certain important special purpose applications.

(4) The feature's inclusion enhances portability.

(5) The feature uses modern language technology.

(6) The feature is compatible with the primary and decremental features.

A.1.3 Decremental Features. Decremental features are those features that are deleted, obsolescent, or deprecated in the Fortran Standard. They are candidates for removal from future versions of the Fortran Standard. Marking a feature as obsolescent or deprecated does not imply its removal from subsequent standards; notification is given that these features *may* be removed in subsequent revisions.

Appendix B further describes decremental features.

A.1.4 Compatibility. All of FORTRAN 77 is included within Fortran 8x. Fortran 8x consists of the complete language of primary, incremental, and decremental features. No segmentation or subsetting of the language is implied. FORTRAN 77 is the combination of the primary features and the decremental features. Programs written in FORTRAN 77 are compatible with Fortran 8x and, with few exceptions, incremental features may be added to existing FORTRAN 77 programs.

FORTRAN FAMILY OF STANDARDS
(Reference Model)

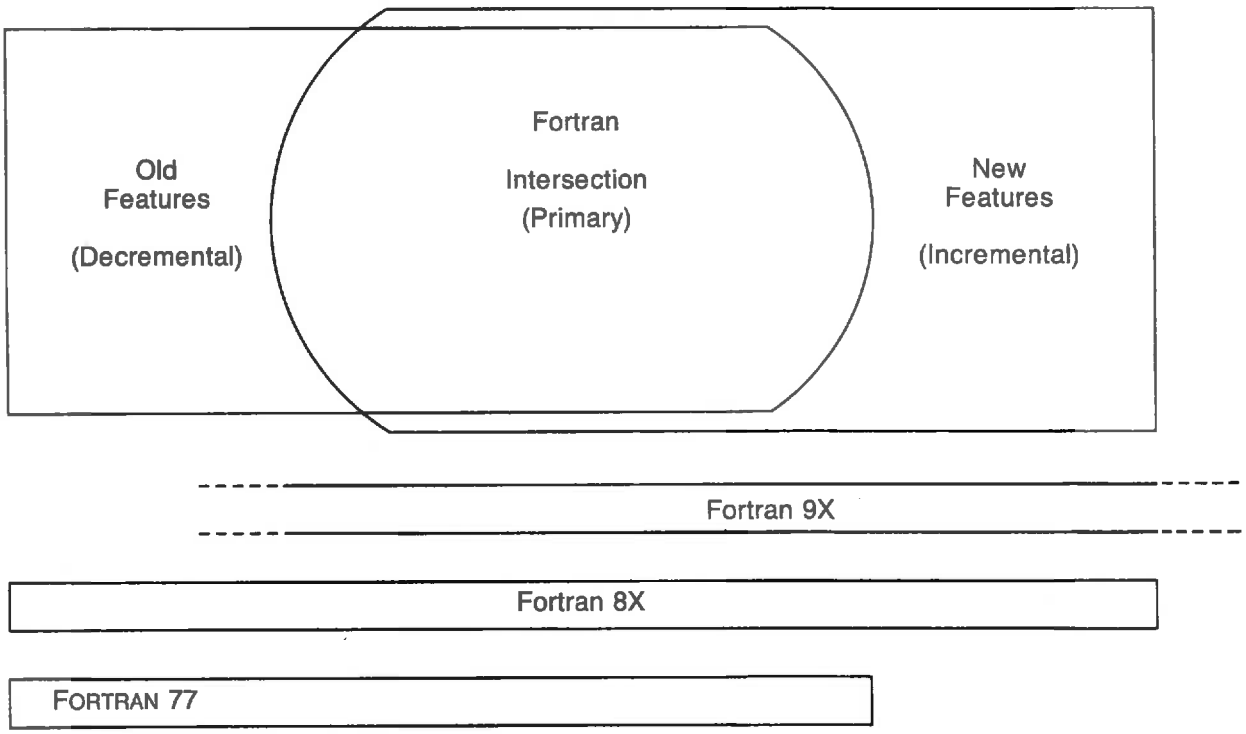


Figure A.1. The Fortran Language Standard.

A.1.5. Core. Core Fortran is the combination of the primary features and incremental features.

A.2. Supplementary Standards Based on Procedure Libraries. Supplementary Standards add functionality to the Fortran language by using the interface mechanisms specified in the Fortran Language Standard. Examples of supplementary standards are the Industrial Real Time Fortran (IRTF) specification and the Fortran binding to the Graphical Kernel System (GKS). These are standards themselves and conform with the FORTRAN 77 standard. Other possible candidates for supplementary standards might be the standardization of certain utility or mathematical libraries and the standardization of data base facilities. While a supplementary standard adds functionality to the Fortran Family, it does not alter the syntax of constructs in Fortran.

A.2.1. Interface Mechanisms. A supplementary standard based on procedure references is called a **procedure supplementary standard**. Such standards must use the interface mechanisms provided in Fortran to describe specific definitions of a process. The interface mechanisms provided in FORTRAN 77 are limited to procedure references. Fortran 8x extends this interface capability by allowing keywords and optional arguments in procedure references.

A.3. Supplementary Standards Based on Module Libraries. A supplementary standard based on modules is called a **module supplementary standard.** Supplementary standards may specify modules that provide a high level of application-oriented functionality. These may include the definition of new data types and their accompanying operators. Modules are nonexecutable program units containing definitions made available to any other program unit by the USE statement. Many problem-oriented applications would make excellent candidates for module supplementary standards. Modules may be included in the Fortran Standard document or they may be standardized in separate documents.

A.3.1. Interface Mechanisms. The interface mechanisms provided in Fortran 8x contain a set of facilities for binding a variety of additional features, such as graphics, to Fortran. These facilities include modules which make definitions, data declarations, and procedure libraries available to an executable program. The USE statement provides the means for referencing specific modules. Supplementary standards may use these mechanisms in defining a specific process within the Fortran Family of Standards.

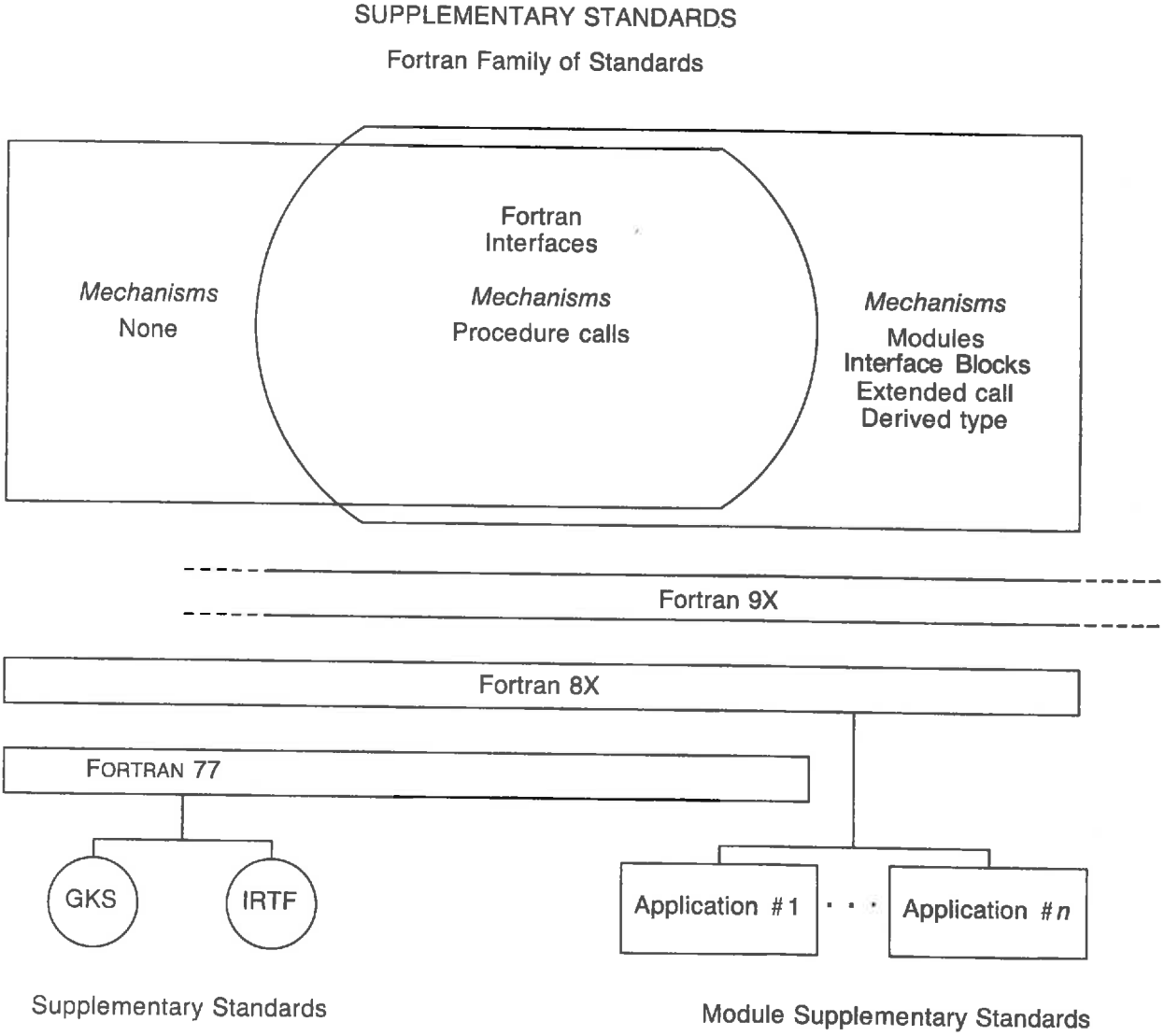


Figure A.2. Supplementary Standards.

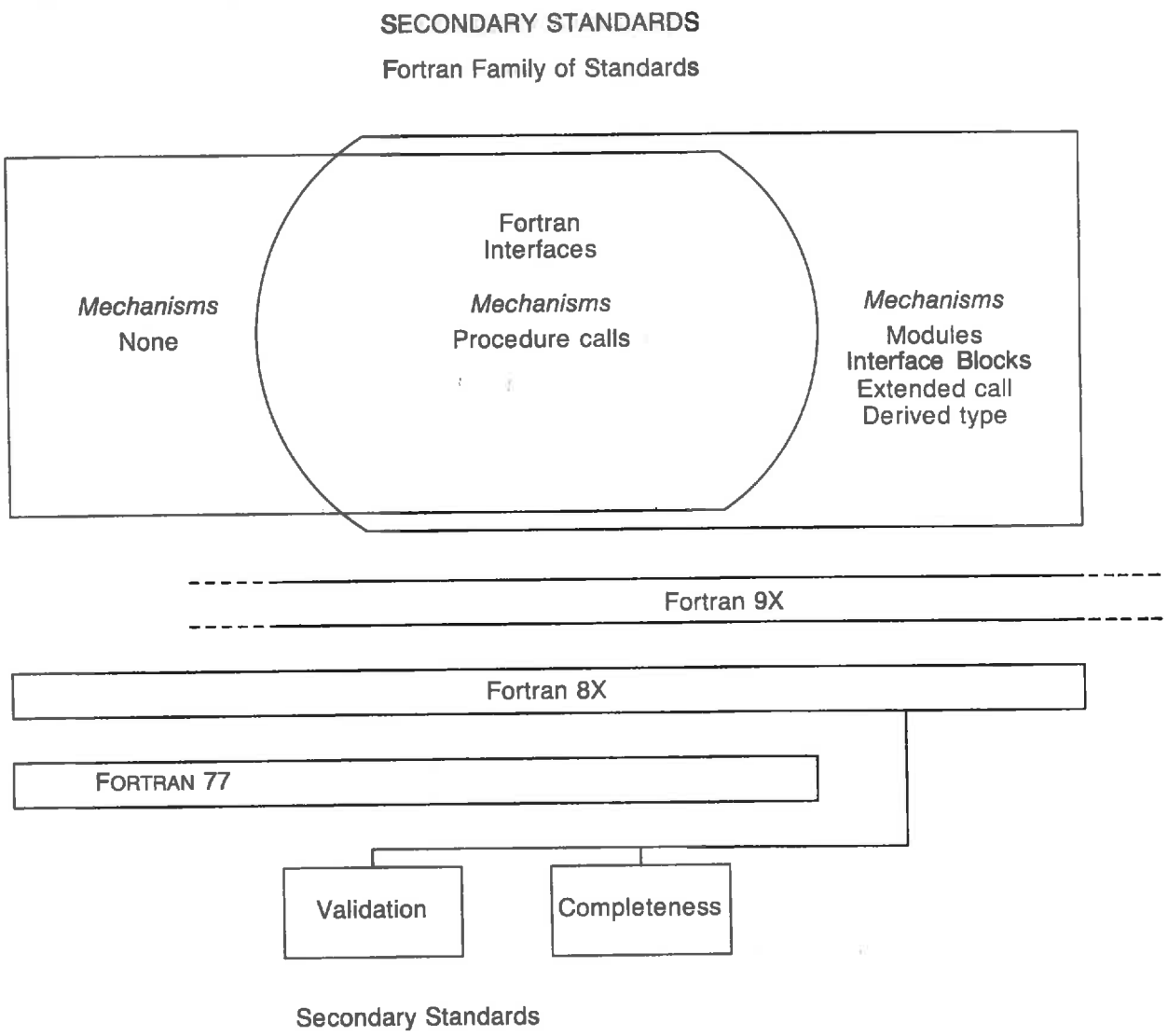


Figure A.3. Secondary Standards.

A.3.2. Rules for Supplementary Standards. Some rules governing the preparation of supplementary standards based on procedure and module libraries are:

(1) A module may be appended to the Fortran Standard or it may be a separate standard.

(2) If a module is appended to the Fortran Standard, it is forwarded for review at the same time as the standard. If it is a separate supplementary standard, there is an independent standardization process.

(3) A module is not part of the Standard. It is a member of the Fortran Family of Standards.

(4) Standard modules must not use obsolescent or deprecated features (i.e., must conform to the Fortran Core.) When the Fortran Standard is revised, a formerly standard-conforming module may cease to be standard conforming because of the use of (old) decremental features.

(5) When the Fortran Standard is revised, a review may determine that modifications are needed to take advantage of any new functionality (incremental features) in the standard.

(6) A name registration for supplementary standards is available from the Fortran Standards Technical Subcommittee.

(7) Separate standards projects should be defined (SD-3) for each supplementary and secondary standard. Task groups may be formed within the Fortran Standards Technical Subcommittee for development of supplementary and secondary standards.

(8) Standard Modules prepared outside the committee and its task groups must use the interface mechanisms in the language. Requests for new facilities in the Fortran Standard must be processed by the Fortran Standards Technical Subcommittee.

(9) The Fortran Standards Technical Subcommittee should review all candidates for supplementary and secondary standards to determine if they are standard conforming. This must be done in a timely manner.

A.4. Secondary Standards. Secondary standards do not impact or change the syntax of the language nor do they change the semantics of the Fortran Standard. Instead, these standards may make requirements on the conformance of programs using the Fortran Standard. For example, certain constructs that control the execution sequence of a program may be required to flag specific conditions that occur during execution. Validation of programs during compilation or execution is another example. Conformance requirements could be expanded in a separate secondary standard. The syntax rules used to help describe the form that Fortran statements take are included in the Fortran Standard (1.5). These rules are described in a variation of BNF. A formal grammar might also be produced as a separate document. Currently, there are no secondary standards in the Fortran Family of Standards; however, work is proceeding in these areas for Fortran and for programming languages in general. See Figure A.3.

A.5. Standard Conformance. Any program unit containing syntax not defined in the Fortran language is not standard conforming with respect to the Fortran Standard. The inclusion of a USE statement does not make the nonstandard conforming syntax standard conforming. A program unit that uses only syntax and semantics defined in the Fortran language standard and one or more standard modules is standard conforming with respect to the Fortran Family of Standards.

In moving to a revised standard, a number of features rather than the complete standard are often selected by implementors. It is recommended that partial implementations of major features not be done. For example, if the array facilities are to be included, as many of the array features as possible should be implemented.

A.5.1. Name Registration. A list of names registered with the Fortran Standards Technical Subcommittee will be kept for reference by those who are preparing a module intended for the Fortran Family of Standards.

A.6. Fortran Family of Standards. Figure A.4 is the complete diagram of the Fortran Family of Standards. It includes the Fortran language with incremental, decremental, and primary features. The interface mechanisms shown refer to the procedure and module supplementary standards in the reference model.

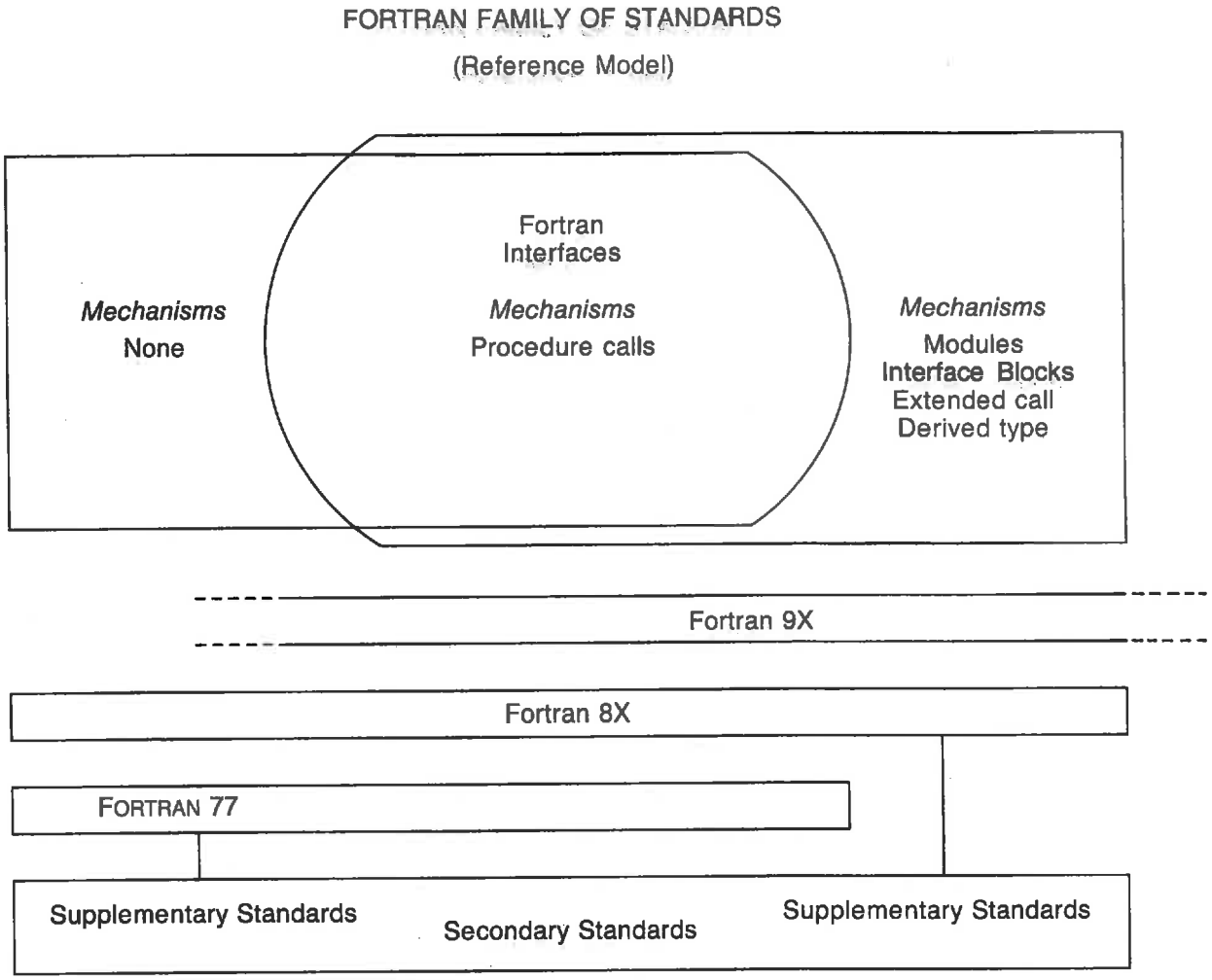


Figure A.4. The Fortran Family of Standards.

APPENDIX B. DECREMENTAL FEATURES

(This appendix is not part of American National Standard X3.9-198x, but is included for information only.)

This appendix more fully describes the rationale for the specific decremental (deleted, obsolescent, and deprecated) features (1.6). Possible alternatives to the obsolescent and deprecated features are described.

B.1. Deleted Features. The deleted features are those features of ANSI X3.9-1978 that are redundant and considered largely unused. Section 1.6.1 describes the nature of the deleted features. The list of deleted features in this standard is empty.

B.2. Obsolescent Features. The obsolescent features are those features of ANSI X3.9-1978 that are redundant and for which better methods are available. Section 1.6.2 describes the nature of obsolescent features. The obsolescent features are:

(1) Arithmetic IF — use a logical IF or a block IF (8.1.2)

(2) Real and double precision DO control variables — use integer (8.1.4.1)

(3) Shared DO termination and termination on a statement other than END DO or CONTINUE — use an END DO or CONTINUE statement for each DO statement

(4) Branching to an END IF statement from outside its IF block — branch to the statement following the END IF

(5) Alternate return — see B.2.1

(6) PAUSE statement — see B.2.2

(7) ASSIGN and assigned GO TO — see B.2.3

(8) Assigned FORMAT specifiers — see B.2.4

B.2.1. Alternate Return. An alternate return introduces a label into an argument list to allow the called subprogram to direct the execution of the caller upon return. Readability and maintainability suffer when alternate returns are used. A better practice is to provide a return code argument that is set by the called subprogram and used in a CASE construct of the calling program unit to direct its subsequent execution. Maintainability is enhanced because an additional case selector may be added without modifying the actual and dummy argument lists. For example:

```
            CALL SUBR_NAME (X, Y, Z, *100, *200, ...)
            ...
100         CONTINUE
            ...
            GO TO 999
200         CONTINUE
            ...
            GO TO 999
            ...
999         CONTINUE
```

where labels 100, 200, etc., are alternate return points. In many cases, the effect can be more safely achieved with a return code and a CASE construct:

```
CALL SUBR_NAME (X, Y, Z, RETURN_CODE)
SELECT CASE (RETURN_CODE)
  CASE (return_1)
    ...
```

```
      CASE (return_2)
        ...
      ...
    END SELECT
```

B.2.2. PAUSE Statement. Execution of a PAUSE statement requires operator or system-specific intervention to resume execution. In most cases, the same functionality can be achieved as effectively and in a more portable way with the use of an appropriate READ statement that awaits some input data.

B.2.3. ASSIGN and Assigned GO TO. The ASSIGN statement allows a label to be dynamically assigned to an integer variable, and the assigned GO TO statement allows "indirect branching" through this variable. This hinders the readability of the program flow, especially if the integer variable also is used in arithmetic operations. The two totally different usages of the integer variable can be an obscure source of error.

Previously, internal subroutines were simulated by the presence of remote code blocks in a procedure. The assigned GO TO statement provided the simulated return from the remote code block "internal subroutine". The addition of internal subroutines to the language replaces this error prone usage.

Example:

```
ASSIGN 120 TO RETURN      ! SET UP RETURN POINT
GO TO 740                 ! BRANCH TO "SUBROUTINE"
120 CONTINUE
...
740 CONTINUE
...                       ! "SUBROUTINE" BODY
GO TO RETURN              ! "SUBROUTINE" RETURN
...
```

This functionality also is provided in this standard through the use of internal subroutines:

```
CALL SUBR_740
...
SUBROUTINE SUBR_740
...                    ! SUBROUTINE BODY
END SUBROUTINE
...
```

This illustrates the use of internal subroutines to conveniently provide "remote code block" functionality. It is believed that more complex uses of ASSIGN and Assigned GO TO can be replaced through the straightforward use of internal procedures, the CASE construct, the IF construct, and if necessary, the unconditional GO TO statement.

B.2.4. Assigned FORMAT Specifiers. The ASSIGN statement also allows the label of a FORMAT statement to be dynamically assigned to an integer variable, which can later be used as a format specifier in READ, WRITE, or PRINT statements. This hinders readability, permits inconsistent usage of the integer variable, and can be an obscure source of error.

This functionality was provided in FORTRAN 77 via character variables, arrays, and constants.

B.3. Nature of Deprecated Features. Section 1.6 describes a set of obsolescent features that are identified in this revision of Fortran. There is another set of features, called the **deprecated features**, which are expected to become obsolescent as the new features of this revision of the Fortran language become widely used. These features are characterized by:

(1) Better methods exist in this document.

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

(3) As the new features of this and subsequent revisions of the standard supplant the deprecated features and they fall into disuse, it is recommended that future Fortran standards committees move these features from the deprecated list to the obsolescent list.

(4) It is recommended that future Fortran standards committees do not consider removing language features defined in this revision that do not exist on this list. (The features in this list should be moved to the obsolescent list as described in (3) before they are considered for deletion.)

(5) It is recommended that processors supporting the Fortran language continue to support these features as long as they continue to be used widely in Fortran programs.

B.3.1 Storage Association. Storage association is the association of data objects through storage sequence patterns rather than by object identification. Storage association allows the user to configure regions of storage and to conserve the use of storage by dynamically designating the objects contained within these storage regions. Though the disadvantages of the use of storage association have been known for some time, features added in this standard have provided Fortran with adequate replacement facilities for important functionality formerly only provided by storage association. The six items below are deprecated due to their use of storage association.

B.3.1.1 Assumed-Size Dummy Arrays. These are dummy arrays declared using an asterisk to specify the last dimension. In this standard, dummy arrays may be declared as assumed-shape arrays by using the colon with no upper bound in one or more dimension positions of the dummy array declaration. Assumed-shape arrays include all of the functionality of assumed-size arrays. Assumed-size arrays assume that a contiguous set of array elements is being passed as an actual argument. With assumed-shape arrays, an array section that does not consist of a contiguous set of array elements (such as a row of a matrix) may also be passed.

B.3.1.2 Passing an Array Element or Substring to a Dummy Array. This functionality is now achieved more safely by passing the desired array section. For example, if a one-dimensional array XX is to be passed starting with the sixth element, then instead of passing XX (6) to the dummy array, one would pass the array section XX (6:); if the eleventh through forty-fifth elements are to be passed, the actual argument would be the array section XX (11:45).

B.3.1.3 BLOCK DATA Program Unit. The principal use of BLOCK DATA program units is to initialize common blocks. Modules provide a complete replacement for BLOCK DATA program units. The global data functionality of common blocks is also provided by modules. Global data in modules may be initialized when specified.

B.3.1.4 COMMON Statement. The important functionality of the COMMON statement has been in its use in specifying global data pools. In this standard, global data pools may be provided more safely and conveniently with MODULE program units and USE statements. Using the COMMON statement, a global data pool could be specified by:

```
INTEGER X (1000)
REAL Y (100, 100)
COMMON / POOL1 / X, Y
```

Each scoping unit using this global data would need to contain these specifications. Alternatively, one can define the global data pool in a MODULE program unit:

```
MODULE POOL1
   INTEGER X (1000)
   REAL Y (100, 100)
END MODULE
```

Each scoping unit using this global data would contain the statement

```
USE POOL1
```

When used in this manner, the MODULE/USE functionality is similar to the INCLUDE extension in many Fortran implementations. This is safer than using common blocks because the specification of the global data pool appears only once. In addition, the USE statement is very short and easy to use. Facilities are provided in the USE statement (not shown here) to rename module objects if different names are desired in the scoping unit using the module objects.

Another advantage is that modules do not involve storage association. Therefore, they may contain any desired mix of character, noncharacter, and derived-type objects. Because a common block involves storage association, a common block cannot contain both character and noncharacter data objects.

B.3.1.5 ENTRY Statement. The ENTRY statement is typically used in situations where there are several operations involving the same set of data objects:

```
procedure-heading
  data-specifications
entry1
  ...
  RETURN
entry2
  ...
  RETURN
...
entryn
  ...
  RETURN
END
```

The MODULE program unit provides the equivalent functionality in the form:

```
MODULE module-name
  data-specifications
procedure-heading1
  ...
  END
procedure-heading2
  ...
  END
...
procedure-headingn
  ...
  END
END MODULE
```

A scoping unit using this module may call each procedure in it, exactly as if they were entry points. One advantage is that some of the subprograms in a module may be functions and some may be subroutines, whereas all entry points in a function subprogram must be invoked as functions and all entry points in a subroutine subprogram must be invoked as subroutines.

B.3.1.6 EQUIVALENCE Statement. A major use of the EQUIVALENCE statement is to have two or more data objects, possibly of different types, share the same storage region. This was important in earlier periods when address space was limited making conservation necessary. The EQUIVALENCE statement also provides the means of simulating certain data types, structures, and transfer functions. This functionality is now available in the language explicitly.

Reuse of storage can now be achieved by using automatic arrays (5.1) and allocatable arrays (5.1.2.4.3). Following the return from the subprogram or deallocation, the space for the dynamic array is available for reuse.

The derived-type capability provides a replacement for the more awkward means of achieving data structures through the use of EQUIVALENCE statements.

The ability of the EQUIVALENCE statement to alias two or more data objects or remap two or more arrays is now provided by the SET RANGE and IDENTIFY statements. Where this new facility is nevertheless inadequate, the TRANSFER function (13.9.7) may be used.

B.3.2 Redundant Functionality. The features identified below are deprecated simply because they are now completely redundant, having been superseded.

(1) Fixed source form — replaced by the free source form (3.3.1)

(2) Specific names for intrinsic functions — use generic names (13.1)

(3) Statement functions — replaced by internal functions (B.3.2.1)

(4) Computed GO TO statement — replaced by CASE construct (see B.3.2.2 below)

(5) The old form of the DATA statement and allowing DATA statements among executable constructs

(6) DIMENSION statement — use type declaration instead (5.1)

(7) DOUBLE PRECISION statement — use precision control attributes (4.3.1.2, 5.1.1.2)

(8) * *char-length* in type specifier — use LEN = *char-length*(5.1.1.5)

B.3.2.1 Use of Internal Functions for Statement Functions. The functionality of the internal function provides a better replacement for the limited statement function capability. For example:

function-name (*dummy-arguments*) = *expr*

may be replaced by the following internal function definition in the internal subprogram part of the program unit.

FUNCTION *function-name* (*dummy-arguments*)
function-and-dummy specifications
function-name = *expr*
END FUNCTION

The use of an internal function in a program unit is the same as the use of a statement function.

B.3.2.2 Example Replacement of the Computed GO TO Statement. The execution sequence controlled by the computed GO TO:

GO TO (*label*$_1$, *label*$_2$, ..., *label*$_n$), *integer-variable*
...
GO TO *label*$_z$

label$_1$ CONTINUE
...
GO TO *label*$_z$

```
        label2 CONTINUE
          ...
          GO TO labelz
        ...
        labeln CONTINUE
          ...
          GO TO labelz
        labelz CONTINUE
```

may be replaced by the CASE construct:

```
        SELECT CASE (integer-variable)
            CASE DEFAULT
                CONTINUE    ! integer-variable < 1 or > n
            CASE (1)
                ...
            CASE (2)
                ...
            CASE (n)
                ...
        END SELECT
```

Also see 8.1.3.

APPENDIX C SECTION NOTES

(This appendix is not part of American National Standard X3.9-198x, but is included for information only.)

C.1 Section 1 Notes. The standard requires a standard-conforming processor to be capable of detecting and reporting the use within a program unit of forms designated as deleted, obsolescent, or deprecated, and of additional forms or relationships, where such use can be detected by reference to the numbered syntax rules and their associated constraints. It is recommended that the processor 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, obsolescent, or deprecated forms. Note that in this context, the use of a deleted form is the use of an additional form.

Use of obsolescent features is discouraged. Each obsolescent feature may be considered for removal in the next revision of the Fortran standard.

C.2 Section 2 Notes. Keywords can make procedure references more readable and allow actual arguments to be in any order. This latter property permits optional arguments.

C.3 Section 3 Notes. A partial collating sequence is specified. If possible, a processor should use the American National Standard Code for Information Interchange, ANSI X3.4-1977 (ASCII), sequence for the complete Fortran character set.

The standard does not restrict the number of consecutive comment lines. The limit of 19 continuation lines or 2640 characters permitted for a statement should not be construed as being a limitation on the number of consecutive comment lines.

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

In fixed source form, an exclamation point (!) 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 another "!" in character positions 1 – 5.

The source form of FORTRAN 77, FORTRAN 66, and the initial Fortran in 1954 was predicated on a common form of input, the 80-column card. However, on the IBM 704, only 72 columns could be used and the remaining eight columns were designated as commentary. In some implementations of FORTRAN 77, these columns are so used. They contain "line numbers" and are used by an editor to manage changes to a program.

In developing Fortran 8x, the Fortran Standards Technical Subcommittee X3J3 sought to eliminate the FORTRAN 77 restriction on source line size. X3J3 believes that 66 positions are inadequate to represent readable Fortran source code, particularly with "long" names and the use of indentation.

Given the need for an incompatible new source form in Fortran 8x, X3J3 relaxed other restrictions of the rigid card form. Positions six and seven are no longer "special" and the continuation mark is on the line being continued rather than on the continuation line. Other features of the Fortran 8x form apply to either form, and are allowed in either.

C.4 Section 4 Notes. A processor must not consider a negative zero to be different from a positive zero.

ANSI X3.9-1978 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 composed of intrinsic types and other 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.

X3.9-1978 provided REAL and DOUBLE PRECISION intrinsic types as approximations to mathematical real numbers. This standard generalizes REAL as an intrinsic type with specifiable precision and exponent range. DOUBLE PRECISION is treated as a synonym for an implementation defined precision and exponent range of the REAL type. Therefore, the DOUBLE PRECISION statement is redundant and use of it is deprecated.

The EXPONENT LETTER statement may be used to designate a letter to be used for the exponent character in real literal constants to ensure that they have a particular precision and exponent range.

X3.9-1978 did not allow zero length character strings. They are permitted by this standard.

Derived types may have parameters as part of the declaration. This allows a derived type to represent simple variations in the data structure such as different string lengths and precisions.

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
COUNT 1 = 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 because they were declared using different derived-type definitions.

The effect of the rules in 4.4.1.1 is to impart a special meaning to and restricted use of the dummy type parameters named PRECISION and EXPONENT_RANGE, and to require (for a given derived type) identical precision and exponent range type parameters for all real, complex, and derived-type components whose precision and exponent range type parameters are specified by dummy type parameters of the derived type. Note that real, complex, or derived-type components whose precision and exponent range type parameters are not specified by dummy type parameters, but by constant expressions, are permitted in any derived type.

A derived type is said to resolve into a sequence of components of intrinsic type. 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 structure in memory as best suited to the particular architecture.

C.5 Section 5 Notes. Type declaration statements in X3.9-1978 required the attributes of an entity to be specified in multiple statements (INTEGER, SAVE, DATA,...). This standard allows most attributes of an entity to be specified in a single extended form of the type statement. For example,

```
INTEGER , ARRAY (10, 10), SAVE :: A, B, C
REAL, PARAMETER :: P1 = 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.

If precision and exponent range are omitted from a REAL declaration, the objects are of default real type. This corresponds to the FORTRAN 77 real type.

The RANGE attribute allows arrays to have a declared upper and lower bound as in FORTRAN 77 and additionally to have a changeable effective lower and upper bound. The effective bounds provide a concise way to set the working bounds on a group of arrays and to improve the readability of the statements. For example, the following statements using the triplet notation

```
A(J:K+1, J-1:K) = B(J:K+1, J-1:K) + C(J:K+1, J-1:K) + C(J:K+1, J:K+1)
A(J:K+1, J-1:K) = A(J:K+1, J-1:K) + A(J:K+1, J-1:K)
```

may be written as follows if the RANGE attribute and SET RANGE statement are used:

```
SET RANGE (J:K+1, J-1:K) A, B, C
A = B + C + C (:,J:K+1)
A = A + A
```

Note that the declared bounds of A, B, and C are not changed by the SET RANGE statement. The only change is to the bounds used when a whole array is referenced or an array section with omitted lower bounds is referenced.

An explicit subscripted reference to an array element outside the effective bounds is allowed and is not an error. Subscript references to elements outside the declared bounds remains undefined as in FORTRAN 77.

C.6 Section 6 Notes. Substrings are of zero length when the starting point exceeds the ending point. This was not allowed in FORTRAN 77. This standard also allows substrings of literal character constants and named character constants.

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
```

The component identified by the reference 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.7. Section 7 Notes. The FORTRAN 77 restriction that none of the character positions being defined in the character assignment statement may be referenced in the expression has been removed (7.5.1.5).

As defined in Section 4, default real and double precision real are described as instances of the real data type. However, for reasons of portability, they are regarded as being different from any instance of the real data type with specified type parameters. In particular, actual arguments of default real type do not associate with dummy arguments of type real with specified type parameters except asterisk. As a result, in order to remain upward compatible with FORTRAN 77, an expression such as R + D, where R is a default real entity and D is a double precision real entity, must be regarded as a double precision real entity, and not an entity of type real with specified type parameters.

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 must not depend on the order of execution. This lack of dependence on order of evaluation permits parallel execution of the function references.

C.8. Section 8 Notes. There are no Section 8 notes.

C.9. Section 9 Notes. What is called a "record" in Fortran is commonly called a "logical record". There is no concept in Fortran of a "physical record".

An endfile record does not necessarily have any physical embodiment. The processor may use a record count or other means to register the position of the file at the time an ENDFILE statement is executed, so that it can take appropriate action when that position is reached again during a read operation. The endfile record, however it is implemented, is considered to exist for the BACKSPACE statement.

This standard accommodates, but does not require, file cataloging. To do this, several concepts are introduced.

Before any input/output can be performed on a file, it must be connected to a unit. The unit then serves as a designator for that file as long as it is connected. To be connected does not imply that "buffers" have or have not been allocated, that "file-control tables" have or have not been filled out, or that any other method of implementation has been used. Connection means that (barring some other fault) a READ or WRITE statement can be executed on the unit, hence on the file. Without a connection, a READ or WRITE statement cannot be executed.

Totally independent of the connection state is the property of existence, this being a file property. The processor "knows" of a set of files that exist at a given time for a given executable program. This set would include tapes ready to read, files in a catalog, a keyboard, a printer, etc. The set may exclude files inaccessible to the executable program because of security, because they are already in use by another executable program, etc. This standard does not specify which files exist, hence wide latitude is available to a processor to implement security, locks, privilege techniques, etc. Existence is a convenient concept to designate all of the files that an executable program can potentially process.

All four combinations of connection and existence may occur:

Connect	Exist	Examples
Yes	Yes	A card reader loaded and ready to be read
Yes	No	A printer before the first line is written
No	Yes	A file named 'JOAN' in the catalog
No	No	A reel of tape destroyed in the fire last week

Means are provided to create, delete, connect, and disconnect files.

A file may have a name. The form of a file name is not specified. If a system does not have some form of cataloging or tape labeling for 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 an executable program. Therefore, this standard does not impose cataloging as a prerequisite. The naming feature is intended to allow use of a cataloging system where one exists.

A file may become connected to a unit in either of two ways: preconnection or execution of an OPEN statement. Preconnection is performed prior to the beginning of execution of an executable program by means external to Fortran. For example, it may be done by job control action or by processor established defaults. Execution of an OPEN statement is not required to access preconnected files.

The OPEN statement provides a means to access existing files that are not preconnected. An OPEN statement may be used in either of two ways: with a file name (open by name) and without a file name (open by unit). A unit is given in either case. Open by name connects

the specified file to the specified unit. Open by unit connects a processor-determined default file to the specified unit. (The default file may or may not have a name.)

Therefore, there are three ways a file may become connected and hence processed: preconnection, open by name, and open by unit. Once a file is connected, there is no means in standard Fortran to determine how it became connected.

An OPEN statement may also be used to create a new file. In fact, any of the foregoing three connection methods may be performed on a file that does not exist. When a unit is preconnected, writing the first record creates the file. With the other two methods, execution of the OPEN statement creates the file.

When a unit becomes connected to a file, either by execution of an OPEN statement or by preconnection, the following connection properties may be established:

(1) An access method, which is sequential or direct, is established for the connection.

(2) A form, which is formatted or unformatted, is established for a connection to a file that exists or is created by the connection. For a connection that results from execution of an OPEN statement, a default form (which depends on the access method, as described in 9.2.1.2) is established if no form is specified. For a preconnected file that exists, a form is established by preconnection. For a preconnected file that does not exist, a form may be established, or the establishment of a form may be delayed until the file is created (for example, by execution of a formatted or unformatted WRITE statement).

(3) A record length may be established. If the access method is direct, the connection established a record length, which specifies the length of each record of the file. An existing file with records that are not all of equal length must 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.

(4) A blank significance property, which is ZERO or NULL, is established for a connection for which the form is formatted. This property has no effect on output. For a connection that results from execution of an OPEN statement, the blank significance property is NULL by default if no blank significance property is specified. For a preconnected file, the property is established by preconnection.

The blank significance property of the connection is effective at the beginning of each formatted input statement. During execution of the statement, any BN or BZ edit descriptors encountered may temporarily change the effect of embedded and trailing blanks.

A processor has wide latitude in adapting these concepts and actions to its own cataloging and job control conventions. Some processors may require job control action to specify the set of files that exist or that will be created by an executable program. Some processors may require no job control action prior to execution. This standard enables processors to perform a dynamic open, close, and file creation, but it does not require such capabilities of the processor.

The meaning of "open" in contexts other that Fortran may include such things as mounting a tape, console messages, spooling, label checking, security checking, etc. These actions may occur upon job control action external to Fortran, upon execution of an OPEN statement, or upon execution of the first read or write of the file. The OPEN statement describes properties of the connection to the file and may or may not cause physical activities to take place. It is a place for an implementation to define properties of a file beyond those required in standard Fortran.

Similarly, the actions of dismounting a tape, protection, etc. of a "close" may be implicit at the end of a run. The CLOSE statement may or may not cause such actions to occur. This is

another place to extend file properties beyond those of standard Fortran. Note, however, that the execution of a CLOSE statement on unit 10 followed by an OPEN statement on the same unit to the same file or to a different file is a permissible sequence of events. The processor must not deny this sequence solely because the implementation chooses to do the physical act of closing the file at the termination of execution of the program.

Table C.1.

Values Assigned to INQUIRE Specifier Variables (assuming no error condition is encountered).

Specifier	INQUIRE by File		INQUIRE by Unit	
	Unconnected	Connected	Connected	Unconnected
EXIST =	.TRUE. if file exists, .FALSE. otherwise		.TRUE. if unit exists, .FALSE. otherwise	
OPENED =	.FALSE.	.TRUE.		.FALSE.
NUMBER =	−1	unit no.		−1
NAMED =	.TRUE. if file named, .FALSE. otherwise			.FALSE.
NAME =	filename (may not be same as FILE = value)		filename if named, else undefined	undefined
ACCESS =	UNDEFINED	SEQUENTIAL or DIRECT		UNDEFINED
SEQUENTIAL =	YES, NO, or UNKNOWN			UNKNOWN
DIRECT =	YES, NO, or UNKNOWN			UNKNOWN
FORM =	UNDEFINED	FORMATTED or UNFORMATTED		UNDEFINED
FORMATTED =	YES, NO, or UNKNOWN			UNKNOWN
UNFORMATTED =	YES, NO, or UNKNOWN			UNKNOWN
RECL =	undefined	if direct access, record length; else undefined		undefined
NEXTREC =	undefined	if direct access, next record #; else undefined		undefined
BLANK =	UNDEFINED	NULL, ZERO, or UNDEFINED		UNDEFINED
DELIM =	UNDEFINED	APOSTROPHE, QUOTE, NONE, or UNDEFINED		UNDEFINED
PAD =	YES	YES or NO		YES
POSITION =	UNDEFINED	REWIND, APPEND, ASIS, or UNDEFINED		UNDEFINED
ACTION =	UNDEFINED	READ, WRITE, or READ/WRITE		UNDEFINED
IOLENGTH =	RECL = value for *output-item-list*			

This standard does not address problems of security, protection, locking, and many other concepts that may be part of the concept of "right of access". Such concepts are considered to be in the province of an operating system.

The OPEN and INQUIRE statements can be extended naturally to consider these things.

Possible access methods for a file are: sequential and direct. The processor may implement two different types of files, each with its own access method. It may also implement one type of file with two different access methods.

Direct access to files is of a simple and commonly available type, that is, fixed-length records. The key is a positive integer.

Keyword forms of specifiers are used because there are many specifiers and a positional notation is difficult to remember. The keyword form sets a style for processor extensions. The UNIT= and FMT= keywords are offered for completeness, but their use is optional. Thus, compatibility with ANSI X3.9-1966 and ANSI X3.9-1978 is achieved.

Format specifications may be included in the READ and WRITE statements, as in:

```
READ (UNIT = 10, FMT = '(I3, A4, F10.2)' ) K, ALPH, X
```

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.

The intent of the VALUES= specifier is to determine, in case of an error or end-of-file condition, how far processing of the input/output list has been completed. In the determination of the value of the VALUES= specifier associated with input/output list items, allowance is made for the expansion of aggregate list items ultimately into equivalent lists of scalar objects, which in formatted input/output are all of intrinsic data types. However, no allowance is made for the correspondence, in some cases, between two values in a record and the matching scalar object of type complex; a count of one for the VALUES= specifier is always associated with such an item for each use. For example, in:

```
COMPLEX :: Z (10)
REAL :: X (10), Y (10)
INTEGER :: IOS, NVALS, I
CHARACTER (LEN = 8) XYZFMT
DATA (XYZFMT = '(6E12.3)')
---
READ (5, XYZFMT, IOSTAT = IOS, VALUES = NVALS) Z
---
READ (5 ,XYZFMT, IOSTAT = IOS, VALUES = NVALS) (X (I), Y (I) ,I = 1, 10)
Z = X + (0.0, 1.0) * Y
```

while both READ statements can process the same external data with functionally equivalent results (if the following assignment is included in the second case), the maximum values with which NVALS is defined are 10 and 20 for the two READ statements respectively.

Allowance is also made for the treatment of a scalar object of a derived type, as a list item in an unformatted input/output statement, as a single, indivisible value.

For example, if STRUCT is a scalar object of a derived type, in:

```
READ (1, IOSTAT = IOS, VALUES = NVALS) STRUCT
```

the maximum value count assignable to NVALS is 1, but in:

```
READ (5, *, IOSTAT = IOS, VALUES = NVALS) STRUCT
```

the value count assigned to NVALS may range anywhere from 0 to the number of scalar objects of intrinsic types into which STRUCT is ultimately resolved.

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.

An example of a restriction on input/output statements (9.8) is that an input statement must not specify that data are to be read from a printer.

C.10. Section 10 Notes. If a character constant is used as a format specifier in an input/output statement, care must be taken that the value of the character constant is a valid format specification. In particular, if the format specification contains an apostrophe edit descriptor, two apostrophes must be written to delimit the apostrophe edit descriptor and four apostrophes must be written for each apostrophe that occurs within the apostrophe edit descriptor. For example, the text:

```
 2 ISN'T 3
```

may be written by various combinations of output statements and format specifications:

```
    WRITE (6, 100) 2, 3
100 FORMAT (1X, I1, 'ISN''T', 1X, I1)

    WRITE (6, '(1X, I1, 1X, ''ISN''''T'', 1X, I1)') 2, 3

    WRITE (6, '(A)') ' 2 ISN''T 3'
```

The T edit descriptor includes the carriage control character in lines that are to be printed. T1 specifies the carriage control character and T2 specifies the first character that is printed.

The length of a formatted record is not always specified exactly and may be processor dependent.

The number of records read by an explicitly formatted input statement can be determined from the following rule: A record is read at the beginning of the format scan (even if the input list is empty), at each slash edit descriptor encountered in the format, and when a format rescan occurs at the end of the format.

The number of records written by an explicitly formatted output statement can be determined from the following rule: A record is written when a slash edit descriptor is encountered in the format, when a format rescan occurs at the end of the format, and at completion of execution of the output statement (even if the output list is empty). Thus, the occurrence of n successive slashes between two other edit descriptors causes $n - 1$ blank lines if the records are printed. The occurrence of n slashes at the beginning or end of a complete format specification causes n blank lines if the records are printed. However, a complete format specification containing n slashes ($n > 0$) and no other edit descriptors causes $n + 1$ blank lines if the records are printed. For example, the statements

```
  PRINT 3
3 FORMAT (/)
```

will write two records that cause two blank lines if the records are printed.

The following examples illustrate list-directed input. A blank character is represented by b.

Example 1:

Program:

```
J = 3
READ *, I
READ *, J
```

Sequential input file;

```
b1b,4bbbbb
,2bbbbbbbb
```

Result: I = 1, J = 3.

Explanation: The second READ statement reads the second record. The initial comma in the record designates a null value; therefore, J is not redefined.

Example 2:

Program:

```
CHARACTER A *8, B *1
READ *, A, B
```

Sequential input file:

```
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 set to the character constant 'bbbbbbbb'. The end of a record acts as a blank, which in this case is a value separator because it occurs between two constants.

C.11. Section 11 Notes. 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 within a computer environment.

A processor may implement an unnamed main program or unnamed block data program unit assigning it a default name. However, this name must not conflict with any other global name in a standard-conforming executable 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.

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. On the other hand, because of the Fortran statement ordering constraints, redundant specifications frequently must 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 must translate redundant specifications of information in multiple program units.

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 USE 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 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
```

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.

The PUBLIC and PRIVATE attributes, which can be declared only in modules, can 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.

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 rather complete data abstraction for a SET data type where the elements of each set are of type integer. The standard set operations of UNION, INTERSECTION, and DIFFERENCE are provided. The CARD function returns the cardinality of (number of elements in) its set argument. Two functions returning logical values are included, ELEMENT and SUBSET, both of which have the operator form .IN.; ELEMENT determines if a given scalar integer value is an element of a given set, and SUBSET determines if a given set is a subset of another given set. (Two sets may be checked for equality by comparing cardinality and checking that one is a subset of the other, or checking to see if each is a subset of the other.)

The transfer function 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. An assignment coercion allows assignment between sets of different sizes, and checks to see if the receiving set data object has an adequate maximum size (returning the null set if not). In this SET implementation, set data objects have a maximum size (number of elements in set) of 200.

```
MODULE INTEGER_SETS

TYPE SET                                    ! DEFINE SET DATA TYPE
   PRIVATE
   INTEGER CARDINAL_NUMBER
   INTEGER ELEMENT_VALUE(200)               ! COULD BE ANY DATA TYPE
END TYPE SET

CONTAINS

INTEGER FUNCTION CARD (A)                   ! RETURNS CARDINALITY OF SET A
   TYPE (SET) A
   CARD = A % CARDINAL_NUMBER
END FUNCTION CARD

LOGICAL FUNCTION ELEMENT (X, A) OPERATOR (.IN.)    ! DETERMINES IF
   INTEGER X                                       ! ELEMENT X IS IN SET A
   TYPE (SET) A
   ELEMENT = .FALSE.
   IF (CARD(A) .EQ. 0) RETURN
   IF (ANY (A % ELEMENT_VALUE (1:CARD(A)) .EQ. X))  ELEMENT = .TRUE.
END FUNCTION ELEMENT
```

```
 1   FUNCTION UNION (A, B)              ! UNION BETWEEN SETS A AND B
 2      TYPE (SET) A, B, UNION
 3      INTEGER J, N
 4      N = CARD (A)
 5      UNION = SETF (A % ELEMENT_VALUE(1:N))
 6      DO J = 1, CARD (B)
 7         IF (.NOT. (B % ELEMENT_VALUE(J) .IN. A)) THEN
 8            N = N+1
 9            UNION % ELEMENT_VALUE(N) = B % ELEMENT_VALUE (J)
10         END IF
11      END DO
12      UNION % CARDINAL_NUMBER = N
13   END FUNCTION UNION

14   FUNCTION DIFFERENCE (A, B)        ! DIFFERENCE OF SETS A AND B
15      TYPE (SET) A, B, DIFFERENCE
16      INTEGER J, X
17      DIFFERENCE = SETF ([1:0])
18      DO J = 1, CARD (A)
19         X = A % ELEMENT_VALUE (J)
20         IF (.NOT. (X .IN. B)) DIFFERENCE = UNION (DIFFERENCE, SETF ([X]))
21      END DO
22   END FUNCTION DIFFERENCE

23   FUNCTION INTERSECTION (A, B)      ! INTERSECTION OF SETS A AND B
24      TYPE (SET) A, B, INTERSECTION
25      INTERSECTION = DIFFERENCE (A, DIFFERENCE (A, B))
26   END FUNCTION INTERSECTION

27   LOGICAL FUNCTION SUBSET (A,B) OPERATOR (.IN.)  ! DETERMINES IF SET A IS
28      TYPE (SET) A, B                             ! A SUBSET OF SET B
29      SUBSET = CARD (A) .LE. CARD (B)             ! OVERLOADS .IN. OPERATION
30      IF (.NOT. SUBSET) RETURN
31      SUBSET = ALL (A % ELEMENT_VALUE (1 : CARD (A)) .IN. B)
32   END FUNCTION SUBSET

33   TYPE (SET) FUNCTION SETF (V)              ! TRANSFER FUNCTION BETWEEN A
34      INTEGER V (:)                          ! CORRESPONDING SET OF ELEMENTS
35      INTEGER J
36      SETF % CARDINAL_NUMBER = 0             ! REMOVING DUPLICATE VALUES
37      DO J = 1, ESIZE (V)
38         IF (.NOT. (V (J) .IN. SETF)) THEN
39            SETF % CARDINAL_NUMBER = SETF % CARDINAL_NUMBER + 1
40            SETF % ELEMENT_VALUE (SETF % CARDINAL NUMBER) = V (J)
41         END IF
42      END DO
43   END FUNCTION SETF

44   FUNCTION VECTOR (A)                       ! TRANSFER THE VALUES OF SET A
45      TYPE (SET) A                           ! INTO A VECTOR OF ASCENDING ORDER
46      INTEGER, ALLOCATABLE :: VECTOR (:)
47      INTEGER I, J, K
48      ALLOCATE (VECTOR (CARD (A)))
49      VECTOR = A % ELEMENT_VALUE (1 : CARD (A))
50      DO I = 1, CARD (A) - 1
51         DO J = 1, CARD (A) - I
52            IF (VECTOR (J+1) .LT. VECTOR (J) THEN
```

```
            K = VECTOR (J); VECTOR (J) = VECTOR (J+1); VECTOR (J+1) = K
          END IF
        END DO
      END DO
END FUNCTION VECTOR

SUBROUTINE SET_ASSIGNMENT_COERCION (A, B) ASSIGNMENT
   TYPE (SET) A, B
   INTEGER N
   A = SETF ([1:0]); N = CARD (B)
   IF (ESIZE (A % ELEMENT_VALUE) .GE. N) A = SETF (B % ELEMENT_VALUE (1:N))
END SUBROUTINE SET_ASSIGNMENT_COERCION

END MODULE INTEGER_SETS
```

Examples of using INTEGER_SETS (A, B, and C are sets; X is an integer variable):

```
IF (CARD (A) .GT. 10) ...       ! CHECK TO SEE IF A HAS MORE THAN 10 ELEMENTS

IF ((X .IN. A) .AND. .NOT. (X .IN. B)) ... ! CHECK FOR X AN ELEMENT OF A BUT NOT OF B

C = UNION (A, INTERSECTION (B, SETF ([1 : 100])))
                          !  C IS THE UNION OF A AND THE
                          !  RESULT OF B INTERSECTED WITH THE INTEGERS 1 TO 100

IF (CARD (INTERSECTION (A, SETF ([2:100:2]))) .GT. 0) ...
                                          ! DOES A HAVE ANY EVEN
                                          ! NUMBERS IN THE RANGE 1:100?

PRINT *, VECTOR (B)      !  PRINT OUT THE ELEMENTS OF SET B, IN ASCENDING ORDER
```

C.12 Section 12 Notes. Of the various types of procedures described in this section, 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 insure that none of these names conflict with any of the names of the external procedures or other globally named entities in a standard-conforming program. For example, this might be done by including in each such added name a character that is not allowed in a standard-conforming name.

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.

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 contained within 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.

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 keyword arguments 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 allocatable 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 must 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 executable 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 executable program. (In implementation terms, the mere presence of a procedure interface block would not generate an external reference in such an implementation.)

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). On the other hand, 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 must 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 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.

One special case of information being made implicitly available through argument association is the use of dummy arguments with precision or exponent range type parameters that are assumed. The use of these dummy arguments has been constrained such that information is available only about the effective attributes of the actual argument, not the declared attributes. Finally, such procedures may not be used as an actual argument. These restrictions allow implementations in which the translation of such a procedure is a collection of procedures, one for each possible representation of the assumed attribute dummy argument, where the representation of the actual argument in a procedure reference is used to determine which procedure in the collection is actually referenced.

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 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 can thus be discarded.

Note that 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, the intent should be INOUT rather than OUT, even if there is no explicit reference to the value of the dummy argument.

Note also that intent INOUT is not equivalent to the default. The argument corresponding to an intent INOUT dummy argument must always be definable, while an argument corresponding to a dummy argument with default intent need be definable only if the dummy argument is actually redefined.

The restrictions on entities associated with dummy arguments are intended to allow a processor to translate a procedure on the assumption that each dummy argument is distinct from any other entity accessible in the procedure. This allows a variety of optimizations in the translation of the procedure, including implementations of argument association in which the value of the actual argument is maintained in a register or in local storage.

This standard does not allow internal procedures to be used as actual arguments, in part to simplify the problem of insuring 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.

C.13 Section 13 Notes.

C.13.1 Summary of Features. This section is a summary of the principal array features.

C.13.1.1 Whole Array Expressions and Assignments. An important extension is that whole array expressions and assignments 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 overload scalar versions having the same name. All arrays in an expression or across an assignment must "conform"; that is, have exactly the same "rank" (number of dimensions) and "shape" (set of lengths in each dimension), but scalars may be included freely and these are interpreted as being broadcast to a conforming array. Expressions are evaluated before any assignment takes place.

C.13.1.2 Array Sections. Whenever whole arrays may be used, it is also possible to use rectangular slices called "sections". For example:

```
A(:, 1:N, 2, 3:1:-1)
```

consists of a subarray containing the whole of the first effective dimension, positions 1 to N of the second dimension, position 2 of the third dimension and positions 1 to 3 in reverse order for the fourth dimension. This is an artificial example chosen to illustrate the different forms. Of course, the most common use is to select a row or column of an array, for example:

```
A (:, J)
```

C.13.1.3 WHERE Statement. 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.13.1.4 Automatic and Allocatable Arrays. 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.13.1.5 Array Constructors. Arrays, and in particular array constants, may be constructed with array constructors exemplified by:

```
[1.0, 3.0, 7.2]
```

which is a rank-one array of size 3,

```
[10[1.3,2.7], 7.1]
```

which is a rank-one array of size 21 and contains [1.3,2.7] repeated 10 times followed by 7.1, and

```
[1:N]
```

which contains the integers 1, 2, ..., N. Only rank-one arrays may be constructed in this way, but higher dimensional arrays may be made from them by means of the intrinsic function RESHAPE.

C.13.1.6. Intrinsic Functions. All of the FORTRAN 77 intrinsic functions and all of the scalar intrinsic functions that have been added to the language have been extended to be applicable to arrays. Each such function is applied element-by-element to produce an array of the same shape. In addition, the following array intrinsics have been added, many of which return array-valued results.

C.13.1.6.1. Vector and Matrix Multiply Functions.

`DOTPRODUCT (VECTOR_A, VECTOR_B)`	Dot product of two arrays
`MATMUL (MATRIX_A, MATRIX-B)`	Matrix multiplication

C.13.1.6.2. Array Reduction Functions.

`ALL (ARRAY, DIM)`	True if all values are true
`ANY (ARRAY, DIM)`	True if any value is true
`COUNT (ARRAY, DIM)`	Number of true elements in an array.
`MAXVAL (ARRAY, DIM, MASK)`	Maximum value in an array
`MINVAL (ARRAY, DIM, MASK)`	Minimum value in an array
`PRODUCT (ARRAY, DIM, MASK)`	Product of array elements
`SUM (ARRAY, DIM, MASK)`	Sum of array elements

C.13.1.6.3. Array Inquiry Functions.

`ALLOCATED (ARRAY)`	Space allocation query
`DLBOUND (ARRAY, DIM)`	Declared lower dimension bounds of an array
`DSHAPE (SOURCE)`	Declared shape of an array or scalar
`DSIZE (ARRAY, DIM)`	Declared total number of array elements
`DUBOUND (ARRAY, DIM)`	Declared upper dimension bounds of an array
`ELBOUND (ARRAY, DIM)`	Effective lower dimension bounds of an array
`ESHAPE (SOURCE)`	Effective shape of an array or scalar
`ESIZE (ARRAY, DIM)`	Effective total number of array elements
`EUBOUND (ARRAY, DIM)`	Effective upper dimension bounds of an array

C.13.1.6.4. Array Construction Functions.

`MERGE (TSOURCE, FSOURCE, MASK)`	Merge under mask
`PACK (ARRAY, MASK, VECTOR)`	Pack an array into a vector under a mask
`RESHAPE (MOLD, SOURCE, PAD, ORDER)`	Reshape an array
`SPREAD (SOURCE, DIM, NCOPIES)`	Replicates an array by adding a dimension
`UNPACK (VECTOR, MASK, FIELD)`	Unpack a vector into an array under a mask

C.13.1.6.5. Array Manipulation Functions.

`CSHIFT (ARRAY, DIM, SHIFT)`	Circular shift
`EOSHIFT (ARRAY, DIM, SHIFT, BOUNDARY)`	End-off shift
`TRANSPOSE (MATRIX)`	Transpose of matrix

C.13.2 Examples. The array features have the potential to simplify the way that almost any array-using program is conceived and written. Many algorithms involving arrays can now be written conveniently as a series of computations with whole arrays.

C.13.2.1 Unconditional Array Computations. At the simplest level, statements such as A = B + C or S= SUM (A) can take the place of entire DO loops. The loops were required to do array addition or to sum all the elements of an array.

Further examples of unconditional operations on arrays that are simple to write are:

matrix multiply	P = MATMUL (Q, R)
largest array element	L = MAXVAL (P)
factorial N	F = PRODUCT ([2:N])

The Fourier sum $F = \sum_{i=1}^{N} a_i \times \cos x_i$ may also be computed without writing a DO loop if one makes use of the element-by-element definition of array expressions as described in Section 7. Thus, we can write

```
F = SUM (A * COS (X)).
```

The successive stages of calculation of F would then involve the arrays:

A	=	[A(1), ..., A(N)]
X	=	[X(1), ..., X(N)]
COS(X)	=	[COS(X(1)), ..., COS(X(N))]
A*COS(X)	=	[A(1) * COS(X(1)), ..., A(N) * COS(X(N))]

The final scalar result is obtained simply by summing the elements of the last of these arrays. Thus, the processor is dealing with arrays at every step of the calculation.

C.13.2.2 Conditional Array Computations. Suppose we wish to compute the Fourier sum in the above example, but to include only those terms $a(i)\cos x(i)$ that satisfy the condition that the coefficient $a(i)$ is less than 0.01 in absolute value. More precisely, we are now interested in evaluating the conditional Fourier sum

$$CF = \sum_{|a_i| < 0.01} a_i \times \cos x_i$$

where the index runs from 1 to N as before.

This can be done using the MASK parameter of the SUM function, which restricts the summation of the elements of the array A * COS(X) to those elements that correspond to true elements of MASK. Clearly, the 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 set an entire array to zero, we may write simply A = 0; but to set only the negative elements to zero, we need to write the conditional assignment

```
WHERE (A .LT. 0)  A = 0
```

The WHERE statement complements ordinary array assignment by providing array assignment to any subset of an array that can be restricted by a logical expression.

In the Ising model described below, the WHERE statement predominates in use over the ordinary array assignment statement.

C.13.2.3. A Simple Program: The Ising Model. The Ising model is a well-known Monte Carlo simulation in 3-dimensional Euclidean space which is useful in certain physical studies. We will consider in some detail how this might be programmed. The model may be described in terms of a logical array of shape N by N 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.13.2.3.1. Problems To Be Solved. Some of the programming problems that we will need to solve in order to translate the Ising model into Fortran statements using entire arrays are:

(1) Counting nearest neighbors that have the same spin;

(2) Providing an array-valued function to return an array of random numbers; and

(3) Determining which gridpoints are to be flipped.

C.13.2.3.2. Solutions in Fortran. The arrays needed are:

```
LOGICAL ISING (N, N, N), FLIPS (N, N, N)
INTEGER ONES (N, N, N), COUNT (N, N, N)
REAL THRESHOLD (N, N, N)
```

The array-valued function needed is:

```
FUNCTION RAND (N, N, N)
REAL RAND (N, N, N)
```

The transition probabilities may be passed across in the array

```
REAL P (6)
```

The first task is to count the number of nearest neighbors of each gridpoint *g* that have the same spin as *g*.

Assuming that ISING is given to us, the statements

```
ONES = 0
WHERE (ISING) ONES = 1
```

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, DIM = 1, SHIFT = -1)  &
       +CSHIFT(ONES, DIM = 1, SHIFT =  1)  &
       +CSHIFT(ONES, DIM = 2, SHIFT = -1)  &
       +CSHIFT(ONES, DIM = 2, SHIFT =  1)  &
       +CSHIFT(ONES, DIM = 3, SHIFT = -1)  &
       +CSHIFT(ONES, DIM = 3, SHIFT =  1)
```

At this point, COUNT contains the count of nearest neighbor up-spins even at the gridpoints where the Ising model has a down-spin. But we want a count of down-spins at those gridpoints, so we correct COUNT at the down (false) points of ISING by writing:

```
WHERE (.NOT. ISING)  COUNT = 6 - COUNT
```

Our object now is to use these counts of what may be called the "like-minded nearest neighbors" to decide which gridpoints are to be flipped. This decision will be recorded as the true elements of an array FLIP. The decision to flip will be based on the use of uniformly distributed random numbers from the interval $0 \leq p < 1$. 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 set to P (4), P (5), and P (6) in order to achieve the desired probabilities of a flip at those points (P (4), P (5), and P (6) are input parameters in the range 0 to 1).

The thresholds are established by the statements:

```
THRESHOLD = 1.0
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.13.2.3.3. The Complete Fortran Subroutine. The complete code, enclosed in a subroutine that performs a sequence of transitions, is as follows:

```
SUBROUTINE TRANSITION (N, ISING, ITERATIONS, P)

LOGICAL ISING (N, N, N), FLIPS (N, N, N)
INTEGER ONES (N, N, N), COUNT (N, N, N)
REAL THRESHOLD (N, N, N), P (6)

!     This interface block is needed to specify
!     that RAND is array-valued.
INTERFACE
    FUNCTION RAND (N)
    REAL RAND (N, N, N)
END INTERFACE
```

```
DO (ITERATIONS) TIMES
   ONES = 0
   WHERE (ISING)  ONES = 1
   COUNT = CSHIFT (ONES, 1, -1) + CSHIFT (ONES, 1, 1) &
         +CSHIFT (ONES, 2, -1) + CSHIFT (ONES, 2, 1) &
         +CSHIFT (ONES, 3, -1) + CSHIFT (ONES, 3, 1)
   WHERE (.NOT. ISING)  COUNT = 6 - COUNT
   THRESHOLD = 1.0
   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
END
```

C.13.2.3.4 Reduction of Storage. The array ISING could be removed (at some loss of clarity) by representing the model in ONES all the time. The array FLIPS can be avoided by combining the two statements that use it as:

```
WHERE (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.13.3 FORmula TRANslation and Array Processing. Many mathematical formulas can be translated directly into Fortran by use of the array processing features.

We assume the following array declarations:

```
REAL X (N), A (M, N)
```

Some examples of mathematical formulas and corresponding Fortran expressions follow.

C.13.3.1 A Sum of Products. The expression

$$\sum_{j=1}^{N} \prod_{i=1}^{M} a_{ij}$$

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.13.3.2 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.13.3.3. Addition of Selected Elements. The expression

$$\sum_{x_i > 0.1} x_i$$

can be formed using the Fortran expression

```
SUM (X, MASK = X .GT. 0.1)
```

The mask locates the elements where the array of rank one is greater than 0.1. If X has the value [0.0, 0.1, 0.2, 0.3, 0.2, 0.1, 0.0], the result of this expression is 0.7.

C.13.4. Variance from the Mean. The expression

$$\sum_{i=1}^{N} (x_i - x_{\text{mean}})^2$$

can be formed using the Fortran statements

```
XMEAN = SUM (X) / ESIZE (X)
VAR = SUM ((X - XMEAN) ** 2)
```

Thus, VAR is the sum of the squared residuals.

C.13.5. 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.13.6. 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.13.7. Logical Queries. The intrinsic functions allow quite complicated questions about tabular data to be answered without use of loops or conditional constructs. Consider, for example, the questions asked below about a simple tabulation of students' test scores.

Suppose the rectangular table T (M, N) contains the test scores of M students who have taken N different tests. T is an integer matrix with entries in the range 0 to 100.

Example: 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) / ESIZE (T); in the example: 62.

Question: How many of the scores in the table are above average?

Answer: ABOVE = T .GT. SUM (T) / ESIZE (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.13.8. Parallel Computations. The most straightforward kind of parallel processing is to do the same thing at the same time to many operands. Matrix addition is a good example of this very simple form of parallel processing. Thus, the array assignment A = B + C specifies that corresponding elements of the identically-shaped arrays B and C be added together in parallel and that the resulting sums be assigned in parallel to the array A.

The "process" being done "in parallel" in the example of matrix addition is of course the process of addition. And the array feature that so successfully implements matrix addition as a parallel process is the element-by-element evaluation of array expressions.

These observations lead us to look to element-by-element computation as a means of implementing other simple parallel processing algorithms.

C.13.9. Examples of Element-by-Element Computation.

C.13.9.1. Polynomials. Several polynomials of the same degree may be evaluated at the same point by arranging their coefficients as the rows of a matrix and applying Horner's method for polynomial evaluation to the columns of the matrix so formed.

This procedure is illustrated by the code to evaluate the three cubic polynomials:

$$P(t) = 1 + 2t - 3t^2 + 4t^3$$

$$Q(t) = 2 - 3t + 4t^2 - 5t^3$$

$$R(t) = 3 + 4t - 5t^2 + 6t^3$$

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.14. Section 14 Notes. There are no Section 14 Notes.

APPENDIX D SYNTAX RULES

(This appendix is not part of American National Standard X3.9-198x, but is included for information only.)

2 FORTRAN TERMS AND CONCEPTS

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

R202 *program-unit* **is** *main-program*
or *external-subprogram*
or *module*
or *block-data*

R203 *main-program* **is** [*program-stmt*]
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-program-stmt

R204 *external-subprogram* **is** *procedure-heading*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
procedure-ending

R205 *procedure-heading* **is** *function-stmt*
or *subroutine-stmt*

R206 *procedure-ending* **is** *end-function-stmt*
or *end-subroutine-stmt*

Constraint: In an *external-subprogram*, *module-subprogram*, or *internal-subprogram*, the *procedure-ending* must be *end-function-stmt* if the *procedure-heading* is a *function-stmt* and must be *end-subroutine-stmt* if the *procedure-heading* is *subroutine-stmt*.

R207 *module* **is** *module-stmt*
[*specification-part*]
[*module-subprogram-part*]
end-module-stmt

Constraint: A module *specification-part* must not contain a *stmt-function-stmt*, an *entry-stmt*, a *format-stmt*, an *intent-stmt*, an INTENT attribute, an *optional-stmt*, or an OPTIONAL attribute.

R208 *block-data* **is** *block-data-stmt*
[*specification-part*]
end-block-data-stmt

Constraint: A *block-data specification-part* may contain only IMPLICIT, PARAMETER, INTEGER, REAL, DOUBLE PRECISION, COMPLEX, CHARACTER, LOGICAL, COMMON, DIMENSION, EQUIVALENCE, DATA, and SAVE statements.

R209 *specification-part* **is** [*use-stmt*]...
[*implicit-part*]
[*declaration-construct*]...
[*stmt-function-part*]

R210 *implicit-part* **is** [*implicit-part-stmt*]...
implicit-stmt

R211 *stmt-function-part* **is** *stmt-function-stmt*
[*stmt-function-part-stmt*]...

R212 *implicit-part-stmt* **is** *implicit-stmt*
or *parameter-stmt*
or *format-stmt*
or *entry-stmt*

R213 *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*

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

R215 *execution-part* **is** *executable-construct*
[*execution-part-construct*]...

R216 *execution-part-construct* **is** *executable-construct*
or *format-stmt*
or *data-stmt*
or *entry-stmt*

R217 *internal-subprogram-part* **is** *contains-stmt*
[*internal-subprogram*]...

R218 *internal-subprogram* **is** *procedure-heading*
[*specification-part*]
[*execution-part*]
procedure-ending

R219 *module-subprogram-part* **is** *contains-stmt*
[*module-subprogram*]...

R220 *module-subprogram* **is** *procedure-heading*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
procedure-ending

R221 *specification-stmt* **is** *access-stmt*
or *data-stmt*
or *exponent-letter-stmt*
or *external-stmt*
or *intent-stmt*
or *intrinsic-stmt*
or *namelist-stmt*

or *optional-stmt*
or *range-stmt*
or *save-stmt*
or *common-stmt*
or *dimension-stmt*
or *equivalence-stmt*

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

Constraint: An *intent-stmt* or *optional-stmt* may appear only in the *specification-part* of a subprogram or the *declaration-construct* of an interface block because they apply only to dummy arguments.

R222 *executable-construct* **is** *action-stmt*
or *case-construct*
or *do-construct*
or *if-construct*
or *where-construct*

R223 *action-stmt* **is** *allocate-stmt*
or *assignment-stmt*
or *backspace-stmt*
or *call-stmt*
or *close-stmt*
or *computed-goto-stmt*
or *continue-stmt*
or *cycle-stmt*
or *deallocate-stmt*
or *endfile-stmt*
or *exit-stmt*
or *goto-stmt*
or *identify-stmt*
or *if-stmt*
or *inquire-stmt*
or *open-stmt*
or *print-stmt*
or *read-stmt*
or *return-stmt*
or *rewind-stmt*
or *set-range-stmt*
or *stop-stmt*
or *where-stmt*
or *write-stmt*
or arithmetic-if-stmt
or assign-stmt
or assigned-goto-stmt
or pause-stmt

Constraint: An *entry-stmt* may appear only in an *external-subprogram* or *module-subprogram*. An *entry-stmt* must not appear within an executable construct.

Constraint: A *return-stmt* may appear only in a subprogram.

Constraint: An *exit-stmt* or a *cycle-stmt* may appear only in a *do-construct*.

3 CHARACTERS, LEXICAL TOKENS, AND SOURCE FORM

R301 *character* **is** *alphanumeric-character*
or *special-character*

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

R303 *underscore* **is** _

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

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

R305 *constant* **is** *literal-constant*
or *named-constant*

R306 *literal-constant* **is** *int-literal-constant*
or *real-literal-constant*
or *complex-literal-constant*
or *logical-literal-constant*
or *char-literal-constant*

R307 *named-constant* **is** *name*

R308 *int-constant* **is** *constant*

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

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

Constraint: *char-constant* must be of type character.

R310 *intrinsic-operator* **is** *power-op*
or *mult-op*
or *add-op*
or *concat-op*
or *rel-op*
or *not-op*
or *and-op*
or *or-op*
or *equiv-op*

R311 *power-op* **is** **

R312 *mult-op* **is** *
or /

R313 *add-op* **is** +
or −

R314 *concat-op* **is** //

R315 *rel-op* **is** .EQ.
or .NE.
or .LT.
or .LE.
or .GT.
or .GE.
or = =
or < >
or <

or < =
or >
or > =

R316 *not-op* **is** .NOT.

R317 *and-op* **is** .AND.

R318 *or-op* **is** .OR.

R319 *equiv-op* **is** .EQV.
or .NEQV.

R320 *defined-operator* **is** *defined-unary-op*
or *defined-binary-op*
or *overloaded-intrinsic-op*

R321 *defined-unary-op* **is** . *letter* [*letter*]... .

R322 *defined-binary-op* **is** . *letter* [*letter*]... .

R323 *overloaded-intrinsic-op* **is** *intrinsic-operator*

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

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

Constraint: At least one digit in the label must be nonzero.

4 INTRINSIC AND DERIVED DATA TYPES

R401 *signed-int-literal-constant* **is** [*sign*] *int-literal-constant*

R402 *int-literal-constant* **is** *digit* [*digit*]...

R403 *sign* **is** +
or –

R404 *signed-real-literal-constant* **is** [*sign*] *real-literal-constant*

R405 *real-literal-constant* **is** *significand* [*exponent-letter exponent*]
or *int-literal-constant exponent-letter exponent*

R406 *significand* **is** *int-literal-constant* . [*int-literal-constant*]
or . *int-literal-constant*

R407 *exponent* **is** *signed-int-literal-constant*

R408 *exponent-letter* **is** E
or D
or *defined-exponent-letter*

R409 *exponent-letter-stmt* **is** EXPONENT LETTER *precision-selector* ■
■ *defined-exponent-letter*

R410 *defined-exponent-letter* **is** *letter*

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

R411 *complex-literal-constant* **is** (*real-part* , *imag-part*)

R412 *real-part* **is** *signed-int-literal-constant*
or *signed-real-literal-constant*

R413 *imag-part* **is** *signed-int-literal-constant*
or *signed-real-literal-constant*

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

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

R416 *derived-type-def* **is** *derived-type-stmt*
[PRIVATE]
component-def-stmt
[*component-def-stmt*]...
end-type-stmt

R417 *derived-type-stmt* **is** [*access-spec*] TYPE *type-name* [(*type-param-name-list*)]

Constraint: A name must not occur more than once in a *type-param-name-list*.

Constraint: If either PRECISION or EXPONENT_RANGE occurs in a *type-param-name-list*, both must occur.

R418 *end-type-stmt* **is** END TYPE [*type-name*]

Constraint: A derived type *type-name* must not be the same as the name of any intrinsic type nor the same as any other accessible derived *type-name*.

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

R419 *component-def-stmt* **is** *type-spec* [[, ARRAY (*explicit-shape-spec-list*)] ::] ■
■ *component-decl-list*

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

Constraint: Each bound in the *explicit-shape-spec* (5.1.2.4.1) must be a nonprecision type-parameter expression (7.1.6.2).

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

R420 *component-decl* **is** *component-name* [(*explicit-shape-spec-list*)] ■
■ [* *char-length*]

Constraint: The * *char-length* option is permitted only if the type specifier is CHARACTER.

Constraint: A *char-length* in a *component-decl* must not contain a *type-param-value* that is an asterisk.

R421 *structure-constructor* **is** *type-name* [(*type-param-spec-list*)] (*expr-list*)

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

R422 *array-constructor* **is** [*array-constructor-value-list*]
or (/ *array-constructor-value-list* /)

R423 *array-constructor-value* **is** *scalar-expr*
or *rank-1-expr*
or *scalar-int-expr* : *scalar-int-expr* [: *constructor-stride*]
or [*scalar-int-expr*] *array-constructor*

R424 *rank-1-expr* **is** *expr*

R425 *constructor-stride* **is** *scalar-int-expr*

Constraint: *rank-1-expr* must have rank one.

5 DATA OBJECT DECLARATIONS AND SPECIFICATIONS

R501 *type-declaration-stmt* **is** *type-spec* [[, *attr-spec*]... ::] *entity-decl-list*

R502 *type-spec*
is INTEGER
or REAL [*precision-selector*]
or DOUBLE PRECISION
or COMPLEX [*precision-selector*]
or CHARACTER [*length-selector*]
or LOGICAL
or TYPE (*type-name* [(*type-param-spec-list*)])

R503 *type-param-spec* **is** [*type-param-name* =] *type-param-value*

R504 *type-param-value*
is *specification-expr*
or *

R505 *attr-spec*
is *value-spec*
or *access-spec*
or ALIAS
or ALLOCATABLE
or ARRAY (*array-spec*)
or INTENT (*intent-spec*)
or OPTIONAL
or RANGE [/ *range-list-name* /]
or SAVE

R506 *entity-decl*
is *object-name* [(*array-spec*)] ■
■ [* char-length] [= *constant-expr*]
or *function-name* [* *char-length*]

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

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

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

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

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

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

Constraint: If the ALIAS attribute is specified, no other attribute except the type and ARRAY may be specified.

Constraint: An array declared with an ALIAS or ALLOCATABLE attribute must be specified with an *array-spec* that is a *deferred-shape-spec-list*.

Constraint: The value, accessibility, ALIAS, and SAVE attributes must not be specified for dummy arguments, functions, or objects in a common block. However, the value attribute DATA may be specified using a DATA statement for objects in a named common block, provided the DATA statement appears in a block data program unit.

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

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

■ [, EXPONENT_RANGE = *type-param-value*])
or (EXPONENT_RANGE = *type-param-value* ■
■ [, PRECISION = *type-param-value*])

Constraint: The *type-param-value* for a precision type parameter must be either a specification expression in which no primary is a reference to a variable except as the argument of the EFFECTIVE_PRECISION function or an asterisk. The *type-param-value* for an exponent range type parameter must be either a specification expression in which no primary is a reference to a variable except as the argument of the EFFECTIVE_EXPONENT_RANGE function or an asterisk.

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

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

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

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

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

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

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

Constraint: The maximum rank is seven.

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

R515 *lower-bound* **is** *scalar-int-expr*

R516 *upper-bound* **is** *scalar-int-expr*

Constraint: An explicit-shape array whose bounds depend on the values of nonconstant expressions must be a dummy argument, a function result, or a local array of a procedure.

Constraint: The bounds in an explicit-shape array declaration must be specification expressions (7.1.6.3).

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

R518 *deferred-shape-spec* **is** :

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

Constraint: An *assumed-size-spec* must not appear in an ARRAY attribute.

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

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

Constraint: An *intent-stmt* may occur only in the scoping unit of a subprogram or an interface block.

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

Constraint: An *optional-stmt* may occur only in the scoping unit of a subprogram or an interface block.

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

Constraint: An *access-stmt* may appear only in the scoping unit of a module or of a derived-type definition contained in a module. If it appears in a derived-type definition, it must be a PRIVATE statement and must not have a *use-name-list*. Only one accessibility statement with an omitted *use-name-list* is permitted in the scoping unit of a module; however, more than one PRIVATE statement may appear if each one is contained in a different scoping unit of either a derived-type definition or a module.

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

R523 *save-stmt* **is** SAVE [[::] *saved-object-list*]

R524 *saved-object* **is** *object-name*
or / *common-block-name* /

Constraint: An *object-name* must not be a dummy argument name, a procedure name, a function result name, an automatic data object, an alias name, a range list name, a namelist group name, or the name of an object in a common block.

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

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

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

R526 *data-stmt* **is** DATA *data-stmt-set* [[,] *data-stmt-set*]...
or DATA (*data-value-def-list*)

R527 *data-stmt-set* **is** *data-stmt-object-list* / *data-stmt-value-list* /

R528 *data-stmt-object* **is** *object-name*
or *array-element*
or *substring*
or *data-implied-do*

Constraint: The *parent-string* in *substring* (R605) must not be a *scalar-constant*.

R529 *data-stmt-value* **is** [*data-stmt-repeat* *] *data-stmt-constant*

R530 *data-stmt-constant* **is** *constant*
or *signed-int-literal-constant*
or *signed-real-literal-constant*

R531 *data-stmt-repeat* **is** *scalar-int-constant*

R532 *data-implied-do* **is** (*data-i-do-object-list*, *data-i-do-variable* = ■
■ *scalar-int-expr*, *scalar-int-expr* [, *scalar-int-expr*])

R533 *data-i-do-object* **is** *array-element*
or *substring*
or *data-implied-do*

R534 *data-i-do-variable* **is** *scalar-int-variable*

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

Constraint: The data statement repeat factor must be positive. If the data statement repeat factor is a named constant, it must have been declared previously in the scoping unit or made accessible by use or host association.

Constraint: A variable whose name is included in a *data-stmt-object-list* or a *data-i-do-object-list* must not be of a derived type, a structure component, a dummy argument, made accessible by use or host association, in a named common block unless the DATA statement is in a block data program unit, in a blank common block, an alias object, a character string with zero length, or a function name. An object whose name is included in either of the above object lists must not be an automatic object, a deferred-shape array, or a zero-sized array.

Constraint: The only variables that may appear in subscripts of the *array-element* in a *data-i-do-object* (R533) are DO variables from the same level or an outer level of the *data-implied-do*. Each such DO variable must appear in some subscript of the *array-element*.

R535 *data-value-def* **is** *variable* = *constant-expr*
or *data-init-implied-do* = *data-init-implied-do-value*

R536 *data-init-implied-do* **is** (*data-init-implied-do-object* , *data-init-implied-do-control*)

R537 *data-init-implied-do-object* **is** *array-element*
or *data-init-implied-do*

R538 *data-init-implied-do-control* **is** *data-i-do-variable* = ■
■ *scalar-int-expr* , *scalar-int-expr* [, *scalar-int-expr*]

R539 *data-init-implied-do-value* **is** *array-constructor*

Constraint: Neither the name of *variable* in *data-value-def* (R535) nor the name of *array-element* in *data-init-implied-do-object* (R537) can be accessible names of the whole or part of dummy arguments, procedures, function results, automatic objects, allocatable arrays, alias objects, or objects in a common block.

Constraint: The only variables that may appear in subscripts of the *array-element* in a *data-init-implied-do-object* (R537) are DO variables from the same level or an outer level of the *data-init-implied-do*. Each such DO variable must appear in some subscript of the *array-element*.

Constraint: Each *data-implied-do* control portion or each *data-init-implied-do-control* must conform to the rules of the DO construct (8.1.4.1), except that the DO variable must be of type integer. The only variables that may appear in *scalar-int-expr* are DO variables from an outer *data-init-implied-do-control.*

Constraint: Each element of the array constructor must be a scalar constant expression.

R540 *parameter-stmt* **is** PARAMETER (*named-constant-def-list*)

R541 *named-constant-def* **is** *named-constant* = *constant-expr*

R542 *range-stmt* **is** RANGE [/ *range-list-name* /] *array-name-list*

Constraint: An array name in *array-name-list* must not be the name of a function procedure.

R543 *implicit-stmt* **is** IMPLICIT *implicit-spec-list*
or IMPLICIT NONE

R544 *implicit-spec* **is** *type-spec* (*letter-spec-list*)

R545 *letter-spec* **is** *letter* [– *letter*]

Constraint: If the minus and second letter appear, the second letter must follow the first letter alphabetically.

R546 *namelist-stmt* **is** NAMELIST / *namelist-group-name* / *namelist-group-object-list* ■
■ [[,] / *namelist-group-name* / *namelist-group-object-list*]...

R547 *namelist-group-object* **is** *variable-name*

Constraint: A *namelist-group-object* must not be an array dummy argument with nonconstant bounds, a variable with assumed parameters, an automatic object, an alias object, or a deferred-shape array.

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

R548 *equivalence-stmt* **is** EQUIVALENCE *equivalence-set-list*

R549 *equivalence-set* **is** (*equivalence-object* , *equivalence-object-list*)

R550 *equivalence-object* **is** *variable-name*
or *array-element*
or *substring*

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

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

R551 *common-stmt* **is** COMMON [/ [*common-block-name*] /]
common-block-object-list
[[,] / [*common-block-name*] /
common-block-object-list]...

R552 *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-lists* within a scoping unit.

Constraint: A *common-block-object* must not be a dummy argument, an object of derived type, an alias object, an allocatable array, an automatic object, an object of real type unless of default real type or double precision real type, an object of complex type unless of default complex type, an array of zero size, a character string of zero length, or a function name.

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

6 USE OF DATA OBJECTS

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

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

R602 *logical-variable* **is** *variable*

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

R603 *char-variable* **is** *variable*

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

R604 *int-variable* **is** *variable*

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

R605 *substring* **is** *parent-string* (*substring-range*)

R606 *parent-string* **is** *scalar-variable-name*
or *array-element*
or *scalar-structure-component*
or *scalar-constant*

R607 *substring-range* **is** [*scalar-int-expr*] : [*scalar-int-expr*]

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

R608 *structure-component* **is** *parent-structure* % *component-name*

R609 *parent-structure* **is** *scalar-variable-name*
or *array-variable-name*
or *array-element*
or *array-section*
or *structure-component*
or *named-constant*

Constraint: *parent-structure* must be of derived type.

R610 *allocate-stmt* **is** ALLOCATE (*array-allocation-list* ■
■ [, STAT = *stat-variable*])

R611 *stat-variable* **is** *scalar-int-variable*

Constraint: The *stat-variable* must not be allocated within the ALLOCATE statement in which it appears.

R612 *array-allocation* **is** *array-name* (*explicit-shape-spec-list*)

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

Constraint: A bound in an *array-allocation explicit-shape-spec* is not restricted to a specification expression, but must not be an expression involving as a primary an array inquiry function (13.9.11) whose argument is any other array in the same ALLOCATE statement.

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

R613 *deallocate-stmt* **is** DEALLOCATE (*array-name-list* ■
■ [, STAT = *stat-variable*])

Constraint: The *stat-variable* must not be deallocated within the same DEALLOCATE statement in which it appears.

R614 *array-element* **is** *parent-array* (*subscript-list*)

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

R615 *array-section* **is** *parent-array* (*section-subscript-list*) [(*substring-range*)]

Constraint: If *substring-range* is present, *parent-array* must be of type character.

Constraint: At least one *section-subscript* must be a *subscript-triplet*.

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

R616 *parent-array* **is** *array-name*
or *structure-component*

Constraint: A *structure-component* may appear only if the component specified is an array.

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

R618 *section-subscript* **is** *subscript*

or *subscript-triplet*

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

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

R621 *set-range-stmt* **is** SET RANGE ([*effective-range-list*]) *array-name-list*
or SET RANGE ([*effective-range-list*]) / *range-list-name* /

R622 *effective-range* **is** *explicit-shape-spec*
or [*lower-bound*] : [*upper-bound*]

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

Constraint: All arrays being ranged must have the same rank and declared lower and upper bounds in corresponding dimensions.

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

R623 *identify-stmt* **is** *identify-scalar-stmt*
or *identify-array-stmt*

R624 *identify-scalar-stmt* **is** IDENTIFY (*scalar-alias-name* = *parent*)

R625 *parent* **is** *scalar-variable-name*
or *array-element*
or *scalar-structure-component*
or *substring*

Constraint: The *scalar-alias-name* and the parent must conform in type and type parameters.

Constraint: The *scalar-alias-name* must have the ALIAS attribute and must not have the SAVE attribute.

Constraint: The *parent* must be a *variable*.

Constraint: The *scalar-alias-name* must not be the same as the name of the scalar parent, either directly or indirectly through multiple alias associations.

R626 *identify-array-stmt* **is** IDENTIFY (*alias-element* = ■
■ *parent-array-element* [(*substring-range*)] , ■
■ *alias-bounds-spec-list*)

R627 *alias-element* **is** *alias-name* (*subscript-name-list*)

R628 *parent-array-element* **is** *parent-array* (*mapping-subscript-list*)

R629 *alias-bound-spec* **is** *subscript-name* = *lower-bound* : *upper-bound*

Constraint: If *substring-range* is present, *parent-array-element* must be of type character. A *subscript-name* must not appear in a *substring-range* in the IDENTIFY statement.

Constraint: The *alias-element* and *parent-array-element* must conform in type and type parameters.

Constraint: Each subscript in a *parent-array-element* must be of a form such that each of the *alias-element subscript-names* appears at most once, and each subscript must be linear in each of the *alias-element subscript-names*. Thus, I+I would not be a valid subscript in a *parent-array-element* where I was an *alias-element subscript-name*. Each of the *alias-element subscript-names* must appear at least once in a *subscript* in the *parent-array-element*.

Constraint: The *alias-element* must have the ALIAS attribute and must not have the SAVE attribute.

Constraint: The *parent-array-element* must be a *variable*.

Constraint: The number of *subscript-names* in an *alias-element* and the number of *alias-bound-specs* must equal the rank of the *alias-name*.

Constraint: Each *subscript-name* in the *subscript-name-list* must be identical to the *subscript-name* in the corresponding *alias-bounds-spec*. The *subscript-names* in both lists must appear in the same order. A *subscript-name* must not appear more than once in each list.

Constraint: A bound in an *alias-bound-spec* must not depend on any other data object or expression in the same IDENTIFY statement nor on any element of the alias object.

Constraint: The alias array elements established by alias association must all be associated with elements of the parent array such that each subscript of a parent array element is within the declared bounds for the corresponding dimension unless the size of that dimension of the alias array is zero.

Constraint: The *alias-element* name must not be the same as the name of the *parent-array-element*, either directly or indirectly through multiple alias associations.

R630 *mapping-subscript* **is** *scalar-int-expr*
or *subscript-map*

R631 *subscript-map* **is** [[*subscript-map*] *add-op*] *subscript-term*

R632 *subscript-term* **is** *add-operand* [* *subscript-name*]
or *subscript-name* [* *subscript-factor*]

R633 *subscript-factor* **is** [*subscript-factor* *] *mult-operand*

Constraint: If a *mapping-subscript* is a *scalar-int-expr*, it must not involve any *subscript-name*. If a *mapping-subscript* is a *subscript-map*, it must involve at least one *subscript-name*.

Constraint: An *add-operand* in a *subscript-term* must not involve any *subscript-name*.

Constraint: A *mult-operand* in a *subscript-factor* must not involve any *subscript-name*.

7 EXPRESSIONS AND ASSIGNMENT

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

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

R321 *defined-unary-op* **is** . *letter* [*letter*]... .

R703 *mult-operand* **is** *level-1-expr* [*power-op mult-operand*]

R704 *add-operand* **is** [*add-operand mult-op*] *mult-operand*

R705 *level-2-expr* **is** [*add-op*] *add-operand*
or *level-2-expr add-op add-operand*

R311 *power-op* **is** **

R312 *mult-op* **is** *
or /

R313 *add-op* **is** +

or –

R706 *level-3-expr* **is** [*level-3-expr concat-op*] *level-2-expr*

R314 *concat-op* **is** //

R707 *level-4-expr* **is** [*level-3-expr rel-op*] *level-3-expr*

R315 *rel-op* **is** .EQ.
or .NE.
or .LT.
or .LE.
or .GT.
or .GE.
or = =
or < >
or <
or < =
or >
or > =

R708 *and-operand* **is** [*not-op*] *level-4-expr*

R709 *or-operand* **is** [*or-operand and-op*] *and-operand*

R710 *equiv-operand* **is** [*equiv-operand or-op*] *or-operand*

R711 *level-5-expr* **is** [*level-5-expr equiv-op*] *equiv-operand*

R316 *not-op* **is** .NOT.

R317 *and-op* **is** .AND.

R318 *or-op* **is** .OR.

R319 *equiv-op* **is** .EQV.
or .NEQV.

R712 *expr* **is** [*expr defined-binary-op*] *level-5-expr*

R322 *defined-binary-op* **is** . *letter* [*letter*]... .

R713 *logical-expr* **is** *expr*

Constraint: *logical-expr* must be type logical.

R714 *char-expr* **is** *expr*

Constraint: *char-expr* must be type character.

R715 *int-expr* **is** *expr*

Constraint: *int-expr* must be type integer.

R716 *numeric-expr* **is** *expr*

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

R717 *constant-expr* **is** *expr*

R718 *char-constant-expr* **is** *char-expr*

R719 *int-constant-expr* **is** *int-expr*

R720 *logical-constant-expr* **is** *logical-expr*

R721 *specification-expr* **is** *scalar-int-expr*

R722 *assignment-stmt* **is** *variable* = *expr*

R723 *where-stmt* **is** WHERE (*mask-expr*) *assignment-stmt*

R724 *where-construct* **is** *where-construct-stmt*
[*assignment-stmt*]...
[*elsewhere-stmt*
[*assignment-stmt*]...]
end-where-stmt

R725 *where-construct-stmt* **is** WHERE (*mask-expr*)

R726 *mask-expr* **is** *logical-expr*

R727 *elsewhere-stmt* **is** ELSEWHERE

R728 *end-where-stmt* **is** END WHERE

Constraint: In each *assignment-stmt*, the *mask-expr* and the variable being defined must be arrays of the same effective shape.

8 EXECUTION CONTROL

R801 *block* **is** [*execution-part-construct*]...

R802 *if-construct* **is** *if-then-stmt*
block
[*else-if-stmt*
block]...
[*else-stmt*
block]
end-if-stmt

R803 *if-then-stmt* **is** [*if-construct-name* :] IF (*scalar-logical-expr*) THEN

R804 *else-if-stmt* **is** ELSE IF (*scalar-logical-expr*) THEN [*if-construct-name*]

R805 *else-stmt* **is** ELSE [*if-construct-name*]

R806 *end-if-stmt* **is** END IF [*if-construct-name*]

Constraint: If an *if-construct-name* is present, the same name must be specified on both the *if-then-stmt* and the corresponding *end-if-stmt*. If an *if-construct-name* appears on the *if-then-stmt*, it may also optionally appear on any *else-if-stmt* or *else-stmt* belonging to that *if-construct*.

R807 *if-stmt* **is** IF (*scalar-logical-expr*) *action-stmt*

Constraint: The *action-stmt* in the *if-stmt* must not be an *if-stmt*.

R808 *case-construct* **is** *select-case-stmt*
[*case-stmt*
block]...
end-select-stmt

R809 *select-case-stmt* **is** [*select-construct-name* :] SELECT CASE (*case-expr*)

R810 *case-stmt* **is** CASE *case-selector* [*select-construct-name*]

R811 *end-select-stmt* **is** END SELECT [*select-construct-name*]

Constraint: If a *select-construct-name* is present, the same name must be specified on both the *select-case-stmt* and the corresponding *end-select-stmt*. If a *select-construct-name* appears on the *select-case-stmt*, it may also optionally appear on any *case-stmt* belonging to that *case-construct*.

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: Only one DEFAULT *case-selector* may appear in any given *case-construct*.

R814 *case-value-range* **is** *case-value*
or *case-value* :
or : *case-value*
or *case-value* : *case-value*

R815 *case-value* **is** *scalar-int-constant-expr*
or *scalar-char-constant-expr*
or *scalar-logical-constant-expr*

Constraint: For a given CASE construct, each *case-value* must be of the same type as *case-expr*. For character type, length differences are allowed.

Constraint: A *case-value-range* using a colon must not be used if *case-expr* is of type logical.

R816 *do-construct* **is** *do-stmt*
do-body
do-termination

R817 *do-stmt* **is** [*do-construct-name* :] DO [*label*] [*loop-control*]

R818 *loop-control* **is** [,] *do-variable* = *scalar-numeric-expr*, ■
■ *scalar-numeric-expr* [, *scalar-numeric-expr*]
or (*scalar-int-expr*) TIMES

R819 *do-variable* **is** *scalar-variable*

Constraint: The *do-variable* must be a scalar integer, default real, or default double precision real named variable.

Constraint: Each *scalar-numeric-expr* in *loop-control* must be of type integer, default real, or default double precision real.

R820 *do-body* **is** [*execution-part-construct*]...

R821 *do-termination* **is** *end-do-stmt*
or *continue-stmt*
or do-term-stmt
or do-construct

R822 *do-term-stmt* is action-stmt

Constraint: If the *label* is omitted in a *do-stmt*, the corresponding *do-termination* must be an *end-do-stmt*.

Constraint: If a *label* appears in the *do-stmt* and the corresponding *do-termination* is not a *do-construct*, the *do-termination* must be identified with that label.

Constraint: If the *do-termination* is a *continue-stmt* or *do-term-stmt*, the corresponding *do-stmt* must contain a label.

Constraint: A *do-term-stmt* must not be a *continue-stmt*, *goto-stmt*, *return-stmt*, *stop-stmt*, *exit-stmt*, *cycle-stmt*, *arithmetic-if-stmt*, nor *assigned-goto-stmt*.

Constraint: If the *do-termination* is a *do-construct*, both of the corresponding *do-stmt*s must specify the same label.

Constraint: If a *do-termination* is a *do-construct*, the *do-termination* of that *do-construct* must not be an *end-do-stmt*.

R823 *end-do-stmt* **is** END DO [*do-construct-name*]

Constraint: If a *do-construct-name* is used on the *do-stmt*, the corresponding *do-termination* must be an *end-do-stmt* that uses the same *do-construct-name*. If a *do-construct-name* does not appear on the *do-stmt*, a *do-construct-name* must not appear on the corresponding *end-do-stmt*.

R824 *exit-stmt* **is** EXIT [*do-construct-name*]

R825 *cycle-stmt* **is** CYCLE [*do-construct-name*]

Constraint: An *exit-stmt* or a *cycle-stmt* must be within the range of one or more *do-constructs*.

Constraint: An *exit-stmt* or *cycle-stmt* using a *do-construct-name* must be within the range of the *do-construct* that has that name.

R826 *goto-stmt* **is** GO TO *label*

Constraint: *label* must be the statement label of a branch target statement that appears in the same scoping unit as the *goto-stmt*.

R827 *computed-goto-stmt* **is** GO TO (*label-list*) [,] *scalar-int-expr*

Constraint: Each *label* in *label-list* must be the statement label of a branch target statement that appears in the same scoping unit as the *computed-goto-stmt*.

R828 *assign-stmt* is ASSIGN *label* TO *scalar-int-variable*

Constraint: *label* must be the statement label of a branch target statement or a *format-stmt*.

R829 *assigned-goto-stmt* is GO TO *scalar-int-variable* [[,] (*label-list*)]

Constraint: Each *label* in *label-list* must be the statement label of a branch target statement that appears in the same scoping unit as the *assigned-goto-stmt*.

R830 *arithmetic-if-stmt* is IF (*scalar-numeric-expr*) *label, label, label*

Constraint: Each *label* must 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* must not be of type complex.

R831 *continue-stmt* **is** CONTINUE

R832 *stop-stmt* **is** STOP [*stop-code*]

R833 *stop-code* **is** *scalar-char-constant*
or *digit* [*digit* [*digit* [*digit* [*digit*]]]]

R834 *pause-stmt* is PAUSE [stop-code]

9 INPUT/OUTPUT STATEMENTS

R901 *io-unit* **is** *external-file-unit*
or *
or *internal-file-unit*

R902 *external-file-unit* **is** *scalar-int-expr*

R903 *internal-file-unit* **is** *char-variable*

R904 *open-stmt* **is** OPEN (*connect-spec-list*)

R905 *connect-spec* **is** [UNIT =] *external-file-unit*
or IOSTAT = *scalar-int-variable*
or ERR = *label*
or FILE = *file-name-expr*
or STATUS = *scalar-char-expr*
or ACCESS = *scalar-char-expr*
or FORM = *scalar-char-expr*
or RECL = *scalar-int-expr*
or BLANK = *scalar-char-expr*

or POSITION = *scalar-char-expr*
or ACTION = *scalar-char-expr*
or DELIM = *scalar-char-expr*
or PAD = *scalar-char-expr*

R906 *file-name-expr* **is** *scalar-char-expr*

Constraint: If the optional characters UNIT = are omitted from the unit specifier, the unit specifier must be the first item in the *connect-spec-list*.

Constraint: Each specifier must not appear more than once in a given *open-stmt*; an *external-file-unit* must be specified.

Constraint: The *label* used in the ERR = specifier must be the statement label of a branch target statement that appears in the same scoping unit as the OPEN statement.

R907 *close-stmt* **is** CLOSE (*close-spec-list*)

R908 *close-spec* **is** [UNIT =] *external-file-unit*
or IOSTAT = *scalar-int-variable*
or ERR = *label*
or STATUS = *scalar-char-expr*

Constraint: If the optional characters UNIT = are omitted from the unit specifier, the unit specifier must be the first item in the *close-spec-list*.

Constraint: The *label* used in the ERR = specifier must be the statement label of a branch target statement that appears in the same scoping unit as the CLOSE statement.

Constraint: A given specifier must not appear more than once in a given *close-stmt*; the unit specifier must appear.

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*]

R912 *io-control-spec* **is** [UNIT =] *io-unit*
or [FMT =] *format*
or [NML =] *namelist-group-name*
or REC = *scalar-int-expr*
or PROMPT = *scalar-char-expr*
or IOSTAT = *scalar-int-variable*
or ERR = *label*
or END = *label*
or NULLS = *scalar-int-variable*
or VALUES = *scalar-int-variable*

Constraint: An *io-control-spec-list* must contain exactly one *io-unit* and may contain at most one of each of the other specifiers.

Constraint: An END = , a NULLS = , or a PROMPT = specifier must not appear in a *write-stmt* or *print-stmt*.

Constraint: The *label* used in the ERR = or END = specifier must 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* must 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* must 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 must be the first item in the control information list.

Constraint: If the optional characters FMT= are omitted from the format specifier, the format specifier must be the second item in the control information list and the first item must be the unit specifier without the optional characters UNIT=.

Constraint: If the optional characters NML= are omitted from the namelist specifier, the namelist specifier must be the second item in the control information list and the first item must be the unit specifier without the optional characters UNIT=.

Constraint: If the unit specifier specifies an internal file, the *io-control-spec-list* must not contain a REC= specifier or a *namelist-group-name*.

Constraint: A NULLS= specifier may be present only in a list-directed input statement.

Constraint: If a *namelist-group-name* is present, a NULLS = specifier must not appear.

Constraint: If a *namelist-group-name* is present, a VALUES= specifier must not appear.

R913 *format* **is** *char-expr*
or *label*
or *
or scalar-int-variable

Constraint: The *label* must be the label of a FORMAT statement that appears in the same scoping unit as the statement containing the format specifier.

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-numeric-expr* , ■
■ *scalar-numeric-expr* [, *scalar-numeric-expr*]

Constraint: The *do-variable* must be scalar of type integer, default real, or double precision real.

Constraint: In an *input-item-list*, an *io-implied-do-object* must be an *input-item*. In an *output-item-list*, an *io-implied-do-object* must be an *output-item*.

R919 *backspace-stmt* **is** BACKSPACE *external-file-unit*
or BACKSPACE (*position-spec-list*)

R920 *endfile-stmt* **is** ENDFILE *external-file-unit*
or ENDFILE (*position-spec-list*)

R921 *rewind-stmt* **is** REWIND *external-file-unit*
or REWIND (*position-spec-list*)

Constraint: BACKSPACE, ENDFILE, and REWIND apply only to external files.

R922 *position-spec* **is** [UNIT =] *external-file-unit*
or IOSTAT = *scalar-int-variable*
or ERR = *label*

Constraint: The *label* used in the ERR= specifier must 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 must be the first item in the *position-spec-list*.

Constraint: A *position-spec-list* must contain exactly one *external-file-unit* and may contain at most one of each of the other specifiers.

R923 *inquire-stmt* **is** INQUIRE (*inquire-spec-list*)
or INQUIRE (IOLENGTH = *scalar-int-variable*) *output-item-list*

R924 *inquire-spec* **is** [UNIT =] *external-file-unit*
or FILE = *file-name-expr*
or IOSTAT = *scalar-int-variable*
or ERR = *label*
or EXIST = *scalar-logical-variable*
or OPENED = *scalar-logical-variable*
or NUMBER = *scalar-int-variable*
or NAMED = *scalar-logical-variable*
or NAME = *scalar-char-variable*
or ACCESS = *scalar-char-variable*
or SEQUENTIAL = *scalar-char-variable*
or DIRECT = *scalar-char-variable*
or FORM = *scalar-char-variable*
or FORMATTED = *scalar-char-variable*
or UNFORMATTED = *scalar-char-variable*
or RECL = *scalar-int-variable*
or NEXTREC = *scalar-int-variable*
or BLANK = *scalar-char-variable*
or POSITION = *scalar-char-variable*
or ACTION = *scalar-char-variable*
or DELIM = *scalar-char-variable*
or PAD = *scalar-char-variable*

Constraint: An INQUIRE statement must contain one FILE= specifier or one UNIT= specifier, but not both, and at most one of each of the other specifiers.

Constraint: In the inquire by unit form of the INQUIRE statement, if the optional characters UNIT= are omitted from the unit specifier, the unit specifier must be the first item in the *inquire-spec-list*.

10 INPUT/OUTPUT EDITING

R1001 *format-stmt* **is** FORMAT *format-specification*

R1002 *format-specification* **is** ([*format-item-list*])

Constraint: The *format-stmt* must be labeled.

Constraint: The comma used to separate *format-items* in a *format-item-list* may be omitted as follows:

(1) Between a P edit descriptor and an immediately following F, E, EN, D, or G edit descriptor (10.6.5)

(2) Before or after a slash edit descriptor when the optional repeat specification is not present (10.6.2)

(3) Before or after a colon edit descriptor (10.6.3)

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* must be positive. It is called a **repeat specification**.

R1005 *data-edit-desc* **is** I *w* [. *m*]
or F *w* . *d*
or E *w* . *d* [E *e*]
or EN *w* . *d* [E *e*]
or G *w* . *d* [E *e*]
or 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: *w* and *e* must be positive and *d* and *m* must be zero or positive.

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*

R1012 *position-edit-desc* **is** T *n*
or TL *n*
or TR *n*
or *n* X

R1013 *n* **is** *int-literal-constant*

Constraint: *n* must be positive.

R1014 *sign-edit-desc* **is** S
or SP
or SS

R1015 *blank-interp-edit-desc* **is** BN
or BZ

R1016 *char-string-edit-desc* **is** *char-literal-constant*
or *c* H *character* [*character*]...

R1017 *c* **is** *int-literal-constant*

Constraint: *c* must be positive.

11 PROGRAM UNITS

R203 *main-program* **is** [*program-stmt*]
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-program-stmt

R1101 *program-stmt* **is** PROGRAM *program-name*

R1102 *end-program-stmt* **is** END [PROGRAM [*program-name*]]

Constraint: In a *main-program*, the *execution-part* must 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, must be identical to the *program-name* specified in the *program-stmt*.

R207 *module* **is** *module-stmt*
[*specification-part*]
[*module-subprogram-part*]
end-module-stmt

R1103 *module-stmt* **is** MODULE *module-name*

R1104 *end-module-stmt* **is** END [MODULE [*module-name*]]

Constraint: If the *module-name* is specified in the *end-module-stmt*, it must be identical to the *module-name* specified in the *module-stmt*.

Constraint: A module *specification-part* must not contain a *stmt-function-stmt*, an *entry-stmt*, a *format-stmt*, an *intent-stmt*, an INTENT attribute, an *optional-stmt*, or an OPTIONAL attribute.

Constraint: An automatic object must not appear in the specification part of a module (R207).

R1105 *use-stmt* **is** USE *module-name* [, *rename-list*]
or USE *module-name* , ONLY : [*only-list*]

R1106 *rename* **is** *use-name* = > *local-name*

R1107 *only* **is** *use-name* [= > *local-name*]

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

R208 *block-data* **is** *block-data-stmt*
[*specification-part*]
end-block-data-stmt

R1108 *block-data-stmt* **is** BLOCK DATA [*block-data-name*]

R1109 *end-block-data-stmt* **is** END [BLOCK DATA [*block-data-name*]]

Constraint: The *block-data-name* may be included in the *end-block-data-stmt* only if it was provided in the *block-data-stmt* and, if included, must be identical to the *block-data-name* in the *block-data-stmt*.

Constraint: A *block-data specification-part* may contain only IMPLICIT, PARAMETER, INTEGER, REAL, DOUBLE PRECISION, COMPLEX, CHARACTER, LOGICAL, COMMON, DIMENSION, EQUIVALENCE, DATA, and SAVE statements.

12 PROCEDURES

R1201 *interface-block* **is** *interface-stmt*
interface-header
[*use-stmt*]...
[*implicit-part*]
[*declaration-construct*]...
end-interface-stmt

R1202 *interface-stmt* **is** INTERFACE

R1203 *end-interface-stmt* **is** END INTERFACE

R1204 *interface-header* **is** *function-stmt*
or *subroutine-stmt*

Constraint: An *interface-block* must not contain an *entry-stmt*.

R1205 *external-stmt* **is** EXTERNAL *external-name-list*

Constraint: Each *external-name* must be the name of an external procedure, a dummy argument, or a block data program unit.

R1206 *intrinsic-stmt* **is** INTRINSIC *intrinsic-procedure-name-list*

R1207 *function-reference* **is** *function-name* ([*actual-arg-spec-list*])

Constraint: The *actual-arg-spec-list* for a function reference must not contain an *alt-return-spec*.

R1208 *call-stmt* **is** CALL *subroutine-name* [([*actual-arg-spec-list*])]

R1209 *actual-arg-spec* **is** [*keyword* =] *actual-arg*

R1210 *keyword* **is** *dummy-arg-name*

R1211 *actual-arg* **is** *expr*
or *variable*
or *procedure-name*
or alt-return-spec

R1212 *alt-return-spec* is * label

Constraint: The *keyword* may be omitted from an *actual-arg-spec* only if the *keyword* has been omitted from each preceding *actual-arg-spec* in the argument list.

Constraint: Each *keyword* must be the name of a dummy argument in the interface of the procedure.

Constraint: A *procedure-name actual-arg* must not be the name of an internal procedure and must not be the name of an intrinsic subroutine (13.8). If it is the name of an intrinsic function, it must be the specific name for the function (13.1).

Constraint: The *label* used in the *alt-return-spec* must be the statement label of a branch target statement that appears in the same scoping unit as the *call-stmt*.

R204 *external-subprogram* **is** *procedure-heading*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
procedure-ending

R205 *procedure-heading* **is** *function-stmt*
or *subroutine-stmt*

R206 *procedure-ending* **is** *end-function-stmt*
or *end-subroutine-stmt*

R1213 *function-stmt* **is** [*prefix*] FUNCTION *function-name* ■
■ ([*dummy-arg-name-list*]) [*suffix*]

R1214 *prefix* **is** *type-spec* [RECURSIVE]
or RECURSIVE [*type-spec*]

R1215 *suffix* **is** RESULT (*result-name*) [OPERATOR (*defined-operator*)]
or OPERATOR (*defined-operator*) [RESULT (*result-name*)]

R1216 *end-function-stmt* **is** END [FUNCTION [*function-name*]]

Constraint: FUNCTION must be present on the *end-function-stmt* of an internal or module function.

Constraint: An internal function must not contain an ENTRY statement.

Constraint: If *function-name* is supplied on the *end-function-stmt*, it must agree with the *function-name* on the *function-stmt*.

R204 *external-subprogram* **is** *procedure-heading*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
procedure-ending

R205 *procedure-heading* **is** *function-stmt*
or *subroutine-stmt*

R206 *procedure-ending* **is** *end-function-stmt*
or *end-subroutine-stmt*

R1217 *subroutine-stmt* **is** [RECURSIVE] SUBROUTINE *subroutine-name* ■
■ [([*dummy-arg-list*])] [ASSIGNMENT]

R1218 *dummy-arg* **is** *dummy-arg-name*
or *

R1219 *end-subroutine-stmt* **is** END [SUBROUTINE [*subroutine-name*]]

Constraint: SUBROUTINE must be present on the END statement of an internal or module subroutine.

Constraint: An internal subroutine must not contain an ENTRY statement.

Constraint: If *subroutine-name* is present on the *end-subroutine-stmt*, it must agree with the *subroutine-name* on the *subroutine-stmt*.

R1220 *entry-stmt* **is** ENTRY *entry-name* [([*dummy-arg-list*])]

Constraint: A *dummy-arg* may be an alternate return indicator only if the ENTRY statement is contained in a subroutine subprogram.

R1221 *return-stmt* **is** RETURN [*scalar-int-expr*]

Constraint: The *return-stmt* must be contained 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.

R1222 *contains-stmt* **is** CONTAINS

R1223 *stmt-function-stmt* **is** *function-name* ([*dummy-arg-name-list*]) = *expr*

Constraint: The *expr* may be composed only of constants (literal and named), references to scalar variables and array elements, references to functions and function dummy procedures, and intrinsic operators. If a reference to another statement function appears in *expr*, its definition must have been provided earlier in the scoping unit.

Constraint: The *function-name* and each *dummy-arg-name* must be specified, explicitly or implicitly, to be scalar data objects.

BNF terms	defined	referenced in
access-spec	R511	R417 R505 R522
access-stmt	R522	R221
action-stmt	R223	R222 R807 R822
actual-arg	R1211	R1209
actual-arg-spec	R1209	R1207 R1208
add-op	R313	R310 R631 R705
add-operand	R704	R632 R704 R705
alias-bounds-spec	R629	R626
alias-element	R627	R626
allocate-stmt	R610	R223
alphanumeric-character	R302	R301 R304
alt-return-spec	R1212	R1211
and-op	R317	R310 R709
and-operand	R708	R709
arithmetic-if-stmt	R830	R223
array-allocation	R612	R610
array-constructor	R422	R423 R539 R701
array-constructor-value	R423	R422
array-element	R614	R528 R533 R537 R550 R601 R606 R609 R625
array-section	R615	R601 R609
array-spec	R513	R505 R506 R525
assign-stmt	R828	R223
assigned-goto-stmt	R829	R223
assignment-stmt	R722	R223 R723 R724
assumed-shape-spec	R517	R513
assumed-size-spec	R519	R513
attr-spec	R505	R501
backspace-stmt	R919	R223
blank-interp-edit-desc	R1015	R1010
block	R801	R802 R808
block-data	R208	R202
block-data-stmt	R1108	R208
c	R1017	R1016
call-stmt	R1208	R223
case-construct	R808	R222
case-expr	R812	R809
case-selector	R813	R810
case-stmt	R810	R808
case-value	R815	R814
case-value-range	R814	R813
char-constant	R309	R833
char-constant-expr	R718	R815
char-expr	R714	R718 R812 R905 R906 R908 R912 R913
char-length	R509	R420 R506 R508
char-literal-constant	R414	R306 R1016
char-string-edit-desc	R1016	R1003
char-variable	R603	R903 R924
character	R301	R414 R1016
close-spec	R908	R907
close-stmt	R907	R223
common-block-object	R552	R551
common-stmt	R551	R221
complex-literal-constant	R411	R306
component-decl	R420	R419
component-def-stmt	R419	R416
computed-goto-stmt	R827	R223
concat-op	R314	R310 R706
connect-spec	R905	R904
constant	R305	R308 R309 R530 R606 R701
constant-expr	R717	R506 R535 R541
constructor-stride	R425	R423
contains-stmt	R1222	R217 R219
continue-stmt	R831	R223 R821
control-edit-desc	R1010	R1003
cycle-stmt	R825	R223
d	R1008	R1005
data-edit-desc	R1005	R1003
data-i-do-object	R533	R532
data-i-do-variable	R534	R532 R538
data-implied-do	R532	R528 R533
data-init-implied-do	R536	R535 R537
data-init-implied-do-control	R538	R536
data-init-implied-do-object	R537	R536
data-init-implied-do-value	R539	R535
data-stmt	R526	R214 R216 R221
data-stmt-constant	R530	R529
data-stmt-object	R528	R527

data-stmt-repeat	R531	R529
data-stmt-set	R527	R526
data-stmt-value	R529	R527
data-value-def	R535	R526
deallocate-stmt	R613	R223
declaration-construct	R213	R209 R1201
deferred-shape-spec	R518	R513
defined-binary-op	R322	R320 R712
defined-exponent-letter	R410	R408 R409
defined-operator	R320	R1215
defined-unary-op	R321	R320 R702
derived-type-def	R416	R213
derived-type-stmt	R417	R416
dimension-stmt	R525	R221
do-body	R820	R816
do-construct	R816	R222 R821
do-stmt	R817	R816
do-term-stmt	R822	R821
do-termination	R821	R816
do-variable	R819	R818 R918
dummy-arg	R1218	R1217 R1220
e	R1009	R1005
effective-range	R622	R621
else-if-stmt	R804	R802
else-stmt	R805	R802
elsewhere-stmt	R727	R724
end-block-data-stmt	R1109	R208
end-do-stmt	R823	R821
end-function-stmt	R1216	R206
end-if-stmt	R806	R802
end-interface-stmt	R1203	R1201
end-module-stmt	R1104	R207
end-program-stmt	R1102	R203
end-select-stmt	R811	R808
end-subroutine-stmt	R1219	R206
end-type-stmt	R418	R416
end-where-stmt	R728	R724
endfile-stmt	R920	R223
entity-decl	R506	R501
entry-stmt	R1220	R212 R213 R214 R216
equiv-op	R319	R310 R711
equiv-operand	R710	R710 R711
equivalence-object	R550	R549
equivalence-set	R549	R548
equivalence-stmt	R548	R221
executable-construct	R222	R215 R216
executable-program	R201	
execution-part	R215	R203 R204 R218 R220
execution-part-construct	R216	R215 R801 R820
exit-stmt	R824	R223
explicit-shape-spec	R514	R419 R420 R513 R519 R552 R612 R622
exponent	R407	R405
exponent-letter	R408	R405
exponent-letter-stmt	R409	R221
expr	R712	R421 R423 R424 R701 R712 R713 R714 R715 R716 R
external-file-unit	R902	R901 R905 R908 R919 R920 R921 R922 R924
external-stmt	R1205	R221
external-subprogram	R204	R202
file-name-expr	R906	R905 R924
format	R913	R909 R911 R912
format-item	R1003	R1002 R1003
format-specification	R1002	R1001
format-stmt	R1001	R212 R213 R214 R216
function-reference	R1207	R701
function-stmt	R1213	R205 R1204
goto-stmt	R826	R223
identify-array-stmt	R626	R623
identify-scalar-stmt	R624	R623
identify-stmt	R623	R223
if-construct	R802	R222
if-stmt	R807	R223
if-then-stmt	R803	R802
imag-part	R413	R411
implicit-part	R210	R209 R1201
implicit-part-stmt	R212	R210
implicit-spec	R544	R543
implicit-stmt	R543	R210 R212

```
input-item                      R914        R909  R917
inquire-spec                    R924        R923
inquire-stmt                    R923        R223
int-constant                    R308        R531
int-constant-expr               R719        R815
int-expr                        R715        R423  R425  R515  R516  R532  R538  R607  R617  R620  R
int-literal-constant            R402        R306  R401  R405  R406  R509  R1004 R1006 R1007 R1008 R
int-variable                    R604        R534  R611  R828  R829  R905  R908  R912  R913  R922  R
intent-spec                     R512        R505  R520
intent-stmt                     R520        R221
interface-block                 R1201       R213
interface-header                R1204       R1201
interface-stmt                  R1202       R1201
internal-file-unit              R903        R901
internal-subprogram             R218        R217
internal-subprogram-part        R217        R203  R204  R220
intrinsic-operator              R310        R323
intrinsic-stmt                  R1206       R221
io-control-spec                 R912        R909  R910
io-implied-do                   R916        R914  R915
io-implied-do-control           R918        R916
io-implied-do-object            R917        R916
io-unit                         R901        R912
k                               R1011       R1010
keyword                         R1210       R1209
label                           R324        R817  R826  R827  R828  R829  R830  R905  R908  R912  R
length-selector                 R508        R502
letter-spec                     R545        R544
level-1-expr                    R702        R703
level-2-expr                    R705        R705  R706
level-3-expr                    R706        R706  R707
level-4-expr                    R707        R708
level-5-expr                    R711        R711  R712
literal-constant                R306        R305
logical-constant-expr           R720        R815
logical-expr                    R713        R720  R726  R803  R804  R807  R812
logical-literal-constant        R415        R306
logical-variable                R602        R924
loop-control                    R818        R817
lower-bound                     R515        R514  R517  R519  R622  R629
m                               R1007       R1005
main-program                    R203        R202
mapping-subscript               R630        R628
mask-expr                       R726        R723  R725
module                          R207        R202
module-stmt                     R1103       R207
module-subprogram               R220        R219
module-subprogram-part          R219        R207
mult-op                         R312        R310  R704
mult-operand                    R703        R633  R703  R704
n                               R1013       R1012
name                            R304        R307
named-constant                  R307        R305  R541  R609
named-constant-def              R541        R540
namelist-group-object           R547        R546
namelist-stmt                   R546        R221
not-op                          R316        R310  R708
numeric-expr                    R716        R818  R830  R918
only                            R1107       R1105
open-stmt                       R904        R223
optional-stmt                   R521        R221
or-op                           R318        R310  R710
or-operand                      R709        R709  R710
output-item                     R915        R910  R911  R917  R923
overloaded-intrinsic-op         R323        R320
parameter-stmt                  R540        R212  R213
parent                          R625        R624
parent-array                    R616        R614  R615  R628
parent-array-element            R628        R626
parent-string                   R606        R605
parent-structure                R609        R608
pause-stmt                      R834        R223
position-edit-desc              R1012       R1010
position-spec                   R922        R919  R920  R921
power-op                        R311        R310  R703
precision-selector              R507        R409  R502
prefix                          R1214       R1213
```

primary	R701	R702
print-stmt	R911	R223
procedure-ending	R206	R204 R218 R220
procedure-heading	R205	R204 R218 R220
program-stmt	R1101	R203
program-unit	R202	R201
r	R1004	R1003 R1010
range-stmt	R542	R221
rank-1-expr	R424	R423
read-stmt	R909	R223
real-literal-constant	R405	R306 R404
real-part	R412	R411
rel-op	R315	R310 R707
rename	R1106	R1105
return-stmt	R1221	R223
rewind-stmt	R921	R223
save-stmt	R523	R221
saved-object	R524	R523
section-subscript	R618	R615
select-case-stmt	R809	R808
set-range-stmt	R621	R223
sign	R403	R401 R404
sign-edit-desc	R1014	R1010
signed-int-literal-constant	R401	R407 R412 R413 R530 R1011
signed-real-literal-constant	R404	R412 R413 R530
significand	R406	R405
specification-expr	R721	R504
specification-part	R209	R203 R204 R207 R208 R218 R220
specification-stmt	R221	R213
stat-variable	R611	R610 R613
stmt-function-part	R211	R209
stmt-function-part-stmt	R214	R211
stmt-function-stmt	R1223	R211 R214
stop-code	R833	R832 R834
stop-stmt	R832	R223
stride	R620	R619
structure-component	R608	R601 R606 R609 R616 R625
structure-constructor	R421	R701
subroutine-stmt	R1217	R205 R1204
subscript	R617	R614 R618 R619
subscript-factor	R633	R632 R633
subscript-map	R631	R630 R631
subscript-term	R632	R631
subscript-triplet	R619	R618
substring	R605	R528 R533 R550 R601 R625
substring-range	R607	R605 R615 R626
suffix	R1215	R1213
type-declaration-stmt	R501	R213
type-param-spec	R503	R421 R502
type-param-value	R504	R503 R507 R508 R509
type-spec	R502	R419 R501 R544 R1214
underscore	R303	R302
upper-bound	R516	R514 R622 R629
use-stmt	R1105	R209 R1201
value-spec	R510	R505
variable	R601	R535 R602 R603 R604 R701 R722 R819 R914 R1211
w	R1006	R1005
where-construct	R724	R222
where-construct-stmt	R725	R724
where-stmt	R723	R223
write-stmt	R910	R223

APPENDIX E PERMUTED INDEX FOR HEADINGS

(This appendix is not part of American National Standard X3.9-198x, but is included for information only.)

APPENDIX F. REMOVED EXTENSIONS

(This appendix is not part of American National Standard X3.9-198x, but is included for information only.)

This appendix contains features that were removed from the draft standard to reduce the size of the language. It is presented for public review and comment. Each of the features described is a possibly valuable extension, but none received sufficient support to be included in the standard.

The additional features include bit data type, variant structures, array features such as vector-valued subscripts and array element assignment (FORALL), procedure extensions such as the nesting of internal procedures, condition handling, and significant blanks in free source form.

F.1. Type Extensions.

F.1.1. Bit Data Type. Bit is a nonnumeric intrinsic type that has two values. Named objects may be declared to be of type BIT and literal constants of type BIT are allowed. Intrinsic operations and functions are provided for objects of this type. Bit objects may appear in expressions and may be used to mask arrays. Bit expressions can appear in control constructs. Input and output is provided for bit objects.

F.1.1.1. Bit Constant. Rule R306 for literal constants must be extended to include a bit constant.

RF01 *literal-constant* **is** *int-literal-constant*
or *real-literal-constant*
or *complex-literal-constant*
or *logical-literal-constant*
or *char-literal-constant*
or *bit-literal-constant*

F.1.1.2. Bit Operators. Rule R310 for intrinsic operators must be extended to include bit operators.

RF02 *intrinsic-operator* **is** *power-op*
or *mult-op*
or *add-op*
or *bnot-op*
or *band-op*
or *bor-op*
or *concat-op*
or *rel-op*
or *not-op*
or *and-op*
or *or-op*
or *equiv-op*

RF03 *bnot-op* **is** .BNOT.

RF04 *band-op* **is** .BAND.

RF05 *bor-op* **is** .BOR.
or .BXOR.

F.1.1.3. Bit Type. The **bit type** has two values which represent the bit values zero and one. Each of these values has two literal representations.

RF06 *bit-literal-constant* **is** B'0'
or B'1'
or B"0"
or B"1"

The values B'0' and B"0" are the same and similarly the values B'1' and B"1" are the same. The intrinsic operations defined for data objects of bit type are: bit negation (.BNOT.), bit conjunction (.BAND.), bit inclusive disjunction (.BOR.), and bit exclusive disjunction (.BXOR.). These operations are defined in F.1.1.5.10. Bit masked array assignment is defined in F.1.1.6.1 and F.1.1.6.2.

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

F.1.1.4. Bit Declaration Statement. A bit object may have rank and shape. There are no additional attributes for objects of type bit. Rule R502 must be extended to include a BIT declaration.

RF07 *type-spec* **is** INTEGER
or REAL [*precision-selector*]
or DOUBLE PRECISION
or COMPLEX [*precision-selector*]
or CHARACTER [*length-selector*]
or LOGICAL
or BIT
or TYPE (*type-name* [(*type-param-spec-list*)])

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

An *equivalence-object* must not be the name of an object of bit type.

A *common-block-object* must not be the name of an object of bit type.

The variables or arrays whose names are included in the *data-i-do-object-list* must not be of type bit.

F.1.1.5. Bit Expressions.

F.1.1.5.1. Bit Objects in Expressions. To include bit expressions, an additional category of expressions is required.

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. For convenience, the low-level operator construction rules, but not the constraints, have been duplicated below from Section 3 where appropriate. See 3.2.4 for the constraints on *defined-unary-op* and *defined-binary-op*. The semantics are specified in 7.1.3 and 7.1.7.

F.1.1.5.2. Primary.

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

Examples of a *primary* are:

Example	Syntactic Class
1.0	*constant*
A	*variable*
[1.0,2.0]	*array-constructor*
PERSON('Jones', 12)	*structure-constructor*
F(X,Y)	*function-reference*
(S + T)	(*expr*)

F.1.1.5.3. Level-1 Expressions. Defined unary operators have the highest operator precedence (Table F.1). Level-1 expressions are primaries optionally operated on by defined unary operators:

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

R321 *defined-unary-op* **is** . *letter* [*letter*]... .

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

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

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

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

F.1.1.5.4. Level-2 Expressions. Level-2 expressions are level-1 expressions optionally involving the numeric operators *power-op*, *mult-op*, and *add-op*.

R703 *mult-operand* **is** *level-1-expr* [*power-op mult-operand*]

R704 *add-operand* **is** [*add-operand mult-op*] *mult-operand*

R705 *level-2-expr* **is** [*add-op*] *add-operand*
or *level-2-expr add-op add-operand*

R311 *power-op* **is** **

R312 *mult-op* **is** *
or /

R313 *add-op* **is** +
or –

Simple examples of a level-2 expression are:

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

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

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

F.1.1.5.5. Level-3 Expressions. Level-3 expressions are level-2 expressions optionally involving the bit operators *bnot-op*, *band-op*, and *bor-op*.

RF08 *band-operand* **is** [*bnot-op*] *level-2-expr*

RF09 *bor-operand* **is** [*bor-operand band-op*] *band-operand*

RF10 *level-3-expr* **is** [*level-3-expr bor-op*] *bor-operand*

RF11 *bnot-op* **is** .BNOT.

RF12 *band-op* **is** .BAND.

RF13 *bor-op* **is** .BOR.
or .BXOR.

Simple examples of a level-3 expression are:

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

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

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

F.1.1.5.6. Level-4 Expressions. Level-4 expressions are level-3 expressions optionally involving the character operator *concat-op*.

RF14 *level-4-expr* **is** [*level-4-expr concat-op*] *level-3-expr*

R314 *concat-op* **is** //

Simple examples of a level-4 expression are:

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

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

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

F.1.1.5.7. Level-5 Expressions. Level-5 expressions are level-4 expressions optionally involving the relational operators *rel-op*.

RF15 *level-5-expr* **is** [*level-4-expr rel-op*] *level-4-expr*

R315 *rel-op* **is** .EQ.
or .NE.
or .LT.
or .LE.
or .GT.
or .GE.
or = =
or < >
or <
or < =
or >
or > =

Simple examples of a level-5 expression are:

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

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

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

F.1.1.5.8. Level-6 Expressions. Level-6 expressions are level-5 expressions optionally involving the logical operators *not-op*, *and-op*, *or-op*, and *equiv-op*.

RF16	*and-operand*	**is** [*not-op*] *level-5-expr*
RF17	*or-operand*	**is** [*or-operand and-op*] *and-operand*
R710	*equiv-operand*	**is** [*equiv-operand or-op*] *or-operand*
RF18	*level-6-expr*	**is** [*level-6-expr equiv-op*] *equiv-operand*
R316	*not-op*	**is** .NOT.
R317	*and-op*	**is** .AND.
R318	*or-op*	**is** .OR.
R319	*equiv-op*	**is** .EQV. **or** .NEQV.

Simple examples of a level-6 expression are:

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

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

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

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

Table F.1. Type of Operands and Result for the Intrinsic Operation $[x_1]$ *op* x_2. (The symbols I, R, Z, B, C, and L stand for the types integer, real, complex, bit, 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.)

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
.BNOT.		B	B
.BAND., .BOR., .BXOR.	B	B	B
//	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
	B	B	L
.GT., .GE., .LT., .LE.	I	I, R	L, L
>, > =, <, < =	R	I, R	L, L
	C	C	L
	B	B	L
.NOT.		L	L
.AND., .OR., .EQV., NEQV.	L	L	L

F.1.1.5.9. Evaluation of Bit Intrinsic Operations. The rules given in F.1.1.5.10 specify the interpretation of bit intrinsic operations. Once the interpretation of an expression has been established in accordance with those rules, the processor may evaluate any other expression that is bit-wise equivalent, provided that the integrity of parentheses is not violated. For example, for variables B1, B2, and B3 of type bit, the processor may choose to evaluate the expression

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

as

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

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

F.1.1.5.10. Bit Intrinsic Operations. A bit operation is used to express a bit computation. Evaluation of a bit operation produces a result of type bit, with a value of B'0' or B'1'. A bit operand may have rank and shape.

The bit operators and their interpretation when used to form an expression are given in Table F.2, where x_1 denotes the operand to the left of the operator and x_2 denotes the operand to the right of the operator.

Table F.2.

Interpretation of the Bit Intrinsic Operators.

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

The values of bit intrinsic operations are shown in Table F.3.

Table F.3.

The Values of Operations Involving Bit Intrinsic Operators

x_1	x_2	.BNOT. x_2	x_1 .BAND. x_2	x_1 .BOR. x_2	x_1 .BXOR. x_2
B'1'	B'1'	B'0'	B'1'	B'1'	B'0'
B'1'	B'0'	B'1'	B'0'	B'1'	B'1'
B'0'	B'1'	B'0'	B'0'	B'1'	B'1'
B'0'	B'0'	B'1'	B'0'	B'0'	B'0'

Derived-type operands may contain bit components.

F.1.1.5.11 Precedence of Bit 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 F.4.

Table F.4.

Categories of Operations and Relative Precedences.

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

The precedence of a defined operation is that of its operator, whether it is an overloaded intrinsic operator or an extension operator.

F.1.1.5.12 Bit Intrinsic Assignment. In Rule R722, *variable* and *expr* may be of type bit provided both are of type bit.

F.1.1.6 Array Mask Expressions of Type Bit. The value of a bit array expression may be used to mask the evaluation of expressions and assignment of values in array assignment statements.

F.1.1.6.1 General Form of the Masked Array Assignment. A **masked array assignment** is either a WHERE statement or WHERE construct.

R723 *where-stmt* **is** WHERE (*mask-expr*) *assignment-stmt*

R724 *where-construct* **is** *where-construct-stmt*
[*assignment-stmt*]...
[*elsewhere-stmt*
[*assignment-stmt*]...]
end-where-stmt

R725 *where-construct-stmt* **is** WHERE (*array-mask-expr*)

RF19 *mask-expr* **is** *logical-expr*
or *bit-expr*

R727 *elsewhere-stmt* **is** ELSEWHERE

R728 *end-where-stmt* **is** END WHERE

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

Examples of a masked array assignment are:

```
WHERE (BIT_ARRAY_TEMP) TEMP = TEMP - REDUCE_TEMP

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

F.1.1.6.2 Interpretation of Masked Array Assignments. The execution of a masked array assignment causes the expression *mask-expr* to be evaluated. The array assignment statements following the WHERE and ELSEWHERE keywords are executed in normal execution sequence. An array may be defined in more than one array assignment statement in a WHERE construct. A reference to an array may appear subsequent to its definition in the same WHERE construct.

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

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

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

F.1.1.7 Bit Expressions in Control Constructs.

F.1.1.7.1 IF Construct. If the scalar mask expression is of type BIT, an expression with value B'1' is treated as true and an expression with value B'0' is treated as false.

F.1.1.7.2 IF Statement. If the scalar mask expression is of type BIT, an expression with value B'1' is treated as true and an expression with value B'0' is treated as false.

F.1.1.7.3 CASE Construct. A case expression may be a scalar bit expression. Rule R812 must be extended.

RF20 *case-expr* **is** *scalar-int-expr*
or *scalar-char-expr*
or *scalar-logical-expr*
or *scalar-bit-expr*

A corresponding case value in a case selector may be a scalar bit constant expression. Rule R815 must be extended.

R814 *case-value-range* **is** *case-value*
or *case-value* :
or : *case-value*
or *case-value* : *case-value*

RF21 *case-value* **is** *scalar-int-constant-expr*
or *scalar-char-constant-expr*
or *scalar-logical-constant-expr*
or *scalar-bit-constant-expr*

If the case value range is of the form *low*: or *:high*, the data type must not be bit.

F.1.1.8 Bit Input/Output Editing.

F.1.1.8.1 Bit Edit Descriptor. There is a bit edit descriptor: B. Rule R1005 must be extended.

RF22 *data-edit-desc* **is** I *w* [. *m*]
or F *w* . *d*
or E *w* . *d* [E *e*]
or EN *w* . *d* [E *e*]
or G *w* . *d* [E *e*]
or B *w*
or L *w*
or A [*w*]
or D *w* . *d*

F.1.1.8.2 B Editing. The B*w* edit descriptor indicates that the field occupies *w* positions. The specified input/output list item must be of type bit.

The input field consists of $w - 1$ blanks and either a 0 or a 1, in any order. The output field consists of $w - 1$ blanks followed by either a 0 or a 1. The specifiers BZ and BN have no effect on bit editing.

F.1.1.8.3 List-Directed and Namelist Output. The form of the bit output constant produced for the value B'1' is 1. The form of the bit output constant produced for the value B'0' is 0.

F.1.1.9 Bit Functions. The elemental functions LBIT and BITL convert between bit and logical type. The transformational functions IBITLR and BITLR convert between a bit array and an integer, counting bits from left to right; IBITRL and BITRL are similar functions that count bits from right to left.

The inquiry function MAXBITS returns the maximum size of a bit array that can be converted to an integer.

BITL (L)	Convert from logical to bit type
BITLR (I, SIZE) Optional SIZE	Convert an integer to a bit array, counting left to right
BITRL (I, SIZE) Optional SIZE	Convert an integer to a bit array, counting right to left
IBITLR (B)	Convert a bit array to an integer, counting left to right
IBITRL (B)	Convert a bit array to an integer, counting right to left
LBIT (B)	Convert from bit to logical type
MAXBITS (I)	Maximum bit array length for conversion

F.1.1.9.1 BITL (L).

Description. Convert logical to bit type.

Kind. Elemental function.

Argument. L must be of type logical.

Result Type. Bit.

Result Value. The result has the value B'1' if L has the value .TRUE. and the value B'0' if L has the value .FALSE.

Example. BITL (.TRUE.) has the value B'1'.

F.1.1.9.2 BITLR (I, SIZE).

Optional Argument. SIZE

Description. Convert an integer to a bit array, counting left to right.

Kind. Transformational function.

Arguments.

I	must be scalar and of type integer. Its value must not be negative.
SIZE (optional)	must be scalar and of type integer with a positive value. If it is omitted, it is as if it were present with the value MAXBITS (1).

Result Type and Shape. The result is a bit array of rank one with SIZE number of elements.

Result Value. The result is a bit array containing the binary representation of the argument. The array element with the largest subscript value will contain the least significant bit of the binary representation. Zero extension or truncation will take place at the low end of the array as necessary. IBITLR (BITLR (J)) must have the value J for any non-negative value of the integer J. BITLR (IBITLR (B), SIZE = ESIZE (B)) must have the value B for any value of a bit array B for which ESIZE (B) $\leq$ MAXBITS (1).

Example. BITLR (5, 6) has the value [B'0', B'0', B'0', B'1', B'0', B'1'].

F.1.1.9.3 BITRL (I, SIZE).

Optional Argument. SIZE

Description. Convert an integer to a bit array, counting right to left.

Kind. Transformational function.

Arguments.

I	must be scalar and of type integer. Its value must not be negative.
SIZE (optional)	must be scalar and of type integer with a positive value. If it is omitted, it is as if it were present with the value MAXBITS (1).

Result Type and Shape. The result is a bit array of rank one with SIZE number of elements.

Result Value. The result is a bit array containing the binary representation of the argument. The array element with the largest subscript value will contain the most significant bit of the binary representation. Zero extension or truncation will take place at the high end of the array as necessary. IBITRL (BITRL (J)) must have the value J for any nonnegative value of the integer J. BITRL (IBITRL (B), SIZE = ESIZE (B)) must have the value B for any value of a bit array B for which ESIZE (B) ≤ MAXBITS (1).

Example. BITRL (5, 6) has the value [B'1', B'0', B'1', B'0', B'0', B'0'].

F.1.1.9.4 IBITLR (B).

Description. Convert a bit array to an integer, counting left to right.

Kind. Transformational function.

Argument. B must be of type bit and rank one. Its size must satisfy the inequality ESIZE (B) ≤ MAXBITS (1).

Result Type and Shape. Scalar integer.

Result Value. The result has value equal to the integer represented by the bits in the array B, regarded as a bit string with the element having the largest subscript value being the least significant bit of the result. IBITLR (BITLR (J)) must have the value J for any nonnegative value of the integer J. BITLR (IBITLR (B), SIZE = ESIZE (B)) must have the value B for any value of a bit array B for which ESIZE (B) ≤ MAXBITS (1).

Example. IBITLR ([B'0', B'1', B'0', B'1']) has the value 5.

F.1.1.9.5 IBITRL (B).

Description. Convert a bit array to an integer, counting right to left.

Kind. Transformational function.

Argument. B must be of type bit and rank one. Its size must satisfy the inequality ESIZE (B) ≤ MAXBITS (1).

Result Type and Shape. Scalar integer.

Result Value. The result has value equal to the integer represented by the bits in the array B, regarded as a bit string with the element having the largest subscript value being the most significant bit of the result. IBITRL (BITRL (J)) must have the value J for any nonnegative value of the integer J. BITRL (IBITRL (B), SIZE = ESIZE (B)) must have the value B for any value of a bit array B for which ESIZE (B) ≤ MAXBITS (1).

Example. IBITRL ([B'1', B'0', B'1', B'0']) has the value 5.

F.1.1.9.6 LBIT (B).

Description. Convert bit to logical type.

Kind. Elemental function.

Argument. B must be of type bit.

Result Type. Logical.

Result Value. The result has the value .TRUE. if B has the value B'1' and the value .FALSE. if B has the value B'0'.

Example. LBIT (B'1') has the value .TRUE.

F.1.1.9.7 MAXBITS (I).

Description. Returns the maximum size of a bit array that can be converted to a value of type integer.

Kind. Inquiry function.

Argument. I must be of type integer.

Result Type and Shape. Integer scalar.

Result Value. The result has value equal to the maximum size of a bit array B that can be converted to integer using IBITLR (B) or IBITRL (B).

F.1.1.10 Bit Mask Argument. An argument named MASK for any intrinsic function (13.12, F.2.4) may be of type bit. When the argument is of type bit, a B'1' value is interpreted as true and a B'0' is interpreted as false. The following intrinsic functions have MASK arguments that may be of type bit: ALL, ANY, COUNT, FIRSTLOC, LASTLOC, MAXLOC, MAXVAL, MERGE, MINLOC, MINVAL, PACK, PRODUCT, PROJECT, SUM, and UNPACK.

F.1.1.11 Bit Storage Sequence. A bit data object has no storage sequence.

F.1.2 Variant Structures.

F.1.2.1 General Form of Variant Structures. Derived data types may contain variant parts. Rule R416 that defines derived types must be extended.

RF23 *derived-type-def* **is** *derived-type-stmt*
[PRIVATE]
component-def-stmt
[*component-def-stmt*]...
[*variant-part*]
end-type-stmt

R417 *derived-type-stmt* **is** [*access-spec*] TYPE *type-name* [(*type-param-name-list*)]

R418 *end-type-stmt* **is** END TYPE [*type-name*]

Constraint: A derived type *type-name* must not be the same as the name of any intrinsic type nor the same as any other accessible derived *type-name*.

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

R419 *component-def-stmt* **is** *type-spec* [[, ARRAY (*explicit-shape-spec-list*)] ::] ■
■ *component-decl-list*

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

R420 *component-decl* **is** *component-name* [(*explicit-shape-spec-list*)] [* *char-length*]

RF24 *variant-part* **is** SELECT CASE (*component-name*)
[*case-stmt*
[*component-def-stmt*]...]...
END SELECT

Constraint: The *component-name* must be the name of the immediately preceding component. It must be scalar, must not lie within a variant part, and must be of type integer, logical, bit, or character.

R810 *case-stmt* **is** CASE *case-selector*

R813 *case-selector* **is** (*case-value-range-list*)
or DEFAULT

Constraint: Only one DEFAULT *case-selector* may appear in any given *case-construct* or *variant-part*.

R814 *case-value-range* **is** *case-value*
or *case-value* :
or : *case-value*
or *case-value* : *case-value*

RF25 *case-value* **is** *scalar-int-constant-expr*
or *scalar-char-constant-expr*
or *scalar-logical-constant-expr*
or *scalar-bit-constant-expr*

Constraint: Each *case-value* must be of the same type as the *component-name* of the SELECT CASE statement.

A variant part specifies alternative sequences of components. Only one such sequence has an interpretation at any given time in a structure of that type. The nonvariant component immediately preceding the variant part of a variant derived type is the **tag component**. It must be scalar and of type integer, logical, bit, or character. The value of the tag component in a structure determines which sequence of components in the varying part is selected. The selection follows the rules for the CASE construct (8.1.3), except that nesting and construct names are prohibited.

An example of a variant structure is:

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

F.1.2.2 Comparison of Entities with Variant Parts. Comparison of entities with variant parts must be defined just as comparison of other entities of derived type must be defined.

F.1.2.3 Definition Status of Variant Structures. When any component of a structure and any other component containing that component becomes undefined, the structure becomes undefined. This does not imply that the undefinition of one component of a structure causes all other components to become undefined. Redefinition or undefinition of the tag name component also causes undefinition of components selected by all cases.

F.1.2.4 Input/Output of Variant Structures. Input/output of variant structures is not specified differently than that for structures with no variant parts. However, the requirement that the format be established prior to any transfer of data (9.4.4) and the possibility of variant components may effectively prevent explicitly formatted (10.1) input to objects of derived types containing variant components, because of the interdependence of the input/output list and the format specification.

F.2 Array Extensions.

F.2.1 Structure Arrays of Arrays Treated as Higher-Order Arrays. Array objects may be of any intrinsic type or derived type.

An array object may be a component or a parent structure that is an element of an array. A resulting data object has array properties if the parent or component has array properties.

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

Example:

```
ARRAY_PARENT % ARRAY_FIELD  !  array component of array parent
```

The IDENTIFY statement (6.2.6) permits the mapping of arrays onto structure arrays of arrays.

F.2.2 Vector-Valued Subscripts. A vector integer expression, used as a subscript, can specify an array section. Rule R618 must be extended:

RF26 *section-subscript* **is** *subscript*
or *subscript-triplet*
or *vector-int-expr*

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

The constraint following rule R616 also must be extended:

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

Each subscript triplet and each rank-one expression in the section subscript list indicates a sequence of subscripts.

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

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

U = [1, 3, 2]
V = [2, 1, 1, 3]

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

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

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

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

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

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

Because Z (3, V) and Z (U, V) contain duplicate elements from Z, the sections Z (3, V) and Z (U, V) must not be redefined as sections.

There are some restrictions on the use of vector-valued subscripts. The left-hand side of an assignment statement (R722) must not include an array element more than once in an array section with vector subscripts. An internal file is a character variable other than an array section with any vector subscripts.

F.2.3 Element Array Assignment—FORALL. The element array assignment statement is used to specify an array assignment in terms of array elements or array sections. The element array assignment may be masked with a scalar logical or bit expression. Rule R223 for *action-stmt* is extended to include the *forall-stmt* and appears as RF40 (F4.3.2).

F.2.3.1 General Form of Element Array Assignment.

RF27 *forall-stmt* **is** FORALL (*forall-triplet-spec-list* [,*scalar-mask-expr*]) ■
▩ *forall-assignment*

RF28 *forall-triplet-spec* **is** *subscript-name* = *subscript* : *subscript* [: *stride*]

Constraint: *subscript-name* must be a *scalar-name* of type integer.

Constraint: A *subscript* or a *stride* in a *forall-triplet-spec* must not contain a reference to any *subscript-name* in the *forall-triplet-spec-list*.

RF29 *forall-assignment* **is** *array-element* = *expr*
or *array-section* = *expr*

Constraint: The *array-section* or *array-element* in a *forall-assignment* must reference all of the *forall-triplet-spec subscript-names*.

For each subscript name in the *forall-assignment*, the set of permitted values is determined on entry to the statement and is

$$m_1 + (k-1) \times m_3, \text{ where } k = 1, 2, \ldots, \text{INT}((m_2 - m_1 + m_3)/m_3)$$

and where m_1, m_2, and m_3 are the values of the first subscript, the second subscript, and the stride respectively in the *forall-triplet-spec*. If *stride* is missing, it is as if it were present with a value of the integer 1. The expression *stride* must not have the value 0. If for some subscript name $INT((m_2 - m_1 + m_3)/m_3) \leq 0$, the *forall-assignment* is not executed.

Examples of element array assignments are:

```
FORALL (I = 1:N, J = 1:N) H (I, J) = 1.0 / REAL (I + J - 1)
```

```
FORALL (I = 1:N, J = 1:N, A (I, J) .NE. 0.0) B (I, J) = 1.0 / A (I, J)
```

F.2.3.2 Interpretation of Element Array Assignments. Execution of an element array assignment consists of the evaluation in any order of the subscript and stride expressions in the *forall-triplet-spec-list*, the evaluation of the scalar mask expression, and the evaluation of the *expr* in the *forall-assignment* for all valid combinations of subscript names for which the scalar mask expression is true, followed by the assignment of these values to the corresponding elements of the array being assigned to. If the scalar mask expression is omitted, it is as if it were present with value true. If the scalar mask expression is of type BIT, an expression with value B'1' is treated as true and an expression value B'0' is treated as false.

The *forall-assignment* must not cause any element of the array being assigned to be assigned a value more than once. The scope of the subscript name is the FORALL statement itself. A function reference appearing in any expression in the *forall-assignment* must not redefine any subscript name.

F.2.4 Intrinsic Functions. Additional array intrinsic functions are provided for array construction (REPLICATE, DIAGONAL), array inquiry (RANK), and array geometric location (FIRSTLOC, LASTLOC, PROJECT).

REPLICATE constructs an array from several copies of an actual argument by increasing the size of one of the dimensions. DIAGONAL constructs a diagonal matrix. PROJECT extracts the elements that lie along an edge of an array. For example, to extract from the integer table TABLE (M, N) the vector containing the first positive number in each column, first locate the desired elements in a logical mask FST (M, N) by:

```
FST = FIRSTLOC (TABLE .GT. 0, DIM = 1)
```

and then assign the elements to FSTC by:

```
FSTC = PROJECT (TABLE, FST, DIM = 1, FIELD = 0)
```

F.2.4.1 DIAGONAL (ARRAY, FILL).

Optional Argument. FILL

Description. Create a diagonal matrix from its diagonal.

Kind. Transformational function.

Arguments.

ARRAY — may be of any type. It must have rank one.

FILL (optional) — must be of the same type and type parameters as ARRAY and must be scalar. It may be omitted for the data types in the following table; in this case it is as if it were present with the value shown.

Type of ARRAY	Value of FILL
Integer	0
Real	0.0
Complex	(0.0, 0.0)
Logical	.FALSE.
Character (*len*)	*len* blanks

Result Type, Type Parameters, and Shape. The result is of the type and type parameters of ARRAY and it has rank two and shape [*n*, *n*] where *n* is the size of ARRAY.

Result Value. Element (*i*, *i*) of the result has value ARRAY (*i*) for $1 \le i \le n$. All other elements have the value FILL.

Example. DIAGONAL ([1, 2, 3]) has the value $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$.

F.2.4.2 PROJECT (ARRAY, MASK, FIELD, DIM).

Optional Argument. DIM

Description. Select masked values from an array.

Kind. Transformational function.

Arguments.

ARRAY — may be of any type. It must not be scalar. Its shape must be defined.

MASK — must be of type logical or bit and of the same shape as ARRAY. If DIM is absent, MASK must have at most one true element; otherwise, each section MASK (s_1, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n) must

have at most one true element.

FIELD must be of the same type and type parameters as ARRAY. It must be scalar if DIM is absent. If DIM is present, FIELD must have rank $n-1$ and shape [E (1:DIM − 1), E (DIM + 1:n)], where E (1:n) is the shape of ARRAY.

DIM (optional) must be scalar and of type integer with value in the range $1 \le \text{DIM} \le n$, where n is the rank of ARRAY.

Result Type, Type Parameters, and Shape. The result is of the type and type parameters of ARRAY. It is scalar if DIM is absent or ARRAY has rank one; otherwise, the result has rank $n-1$ and shape [E (1:DIM − 1), E (DIM + 1:n)] where E (1:n) is the shape of ARRAY.

Result Value.

Case (i): The result of PROJECT (ARRAY, MASK, FIELD) is the element of ARRAY corresponding to the true element of MASK if there is one and is FIELD otherwise. Note that if MASK has zero size, the result has value FIELD.

Case (ii): If ARRAY has rank one, PROJECT (ARRAY, MASK, FIELD, DIM) has value equal to that of PROJECT (ARRAY, MASK, FIELD). Otherwise, the value of element (s_1, ..., $s_{\text{DIM}-1}$, $s_{\text{DIM}+1}$, ..., s_n) of PROJECT (ARRAY, MASK, FIELD, DIM) is equal to PROJECT (ARRAY (s_1, ..., $s_{\text{DIM}-1}$, :, $s_{\text{DIM}+1}$, ..., s_n), MASK (s_1, ..., $s_{\text{DIM}-1}$, :, $s_{\text{DIM}+1}$, ..., s_n), FIELD (s_1, ..., $s_{\text{DIM}-1}$, $s_{\text{DIM}+1}$, ..., s_n)). Note that if ARRAY (and MASK) have zero size because E (DIM) has value zero, the result may have nonzero size with all its values coming from FIELD.

Examples.

Case (i): If V is the array [1, 2, 3, 4] and P is the mask [., ., T, .], where "T" represents .TRUE. and "." represents .FALSE., the value of PROJECT (V, MASK = P, FIELD = 0) is the scalar 3, and the value of PROJECT (V, MASK = V.GT.5, FIELD = 99) is the scalar 99. If A is the array $\begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix}$ and L is the array $\begin{bmatrix} . & . & . & . \\ . & . & T & . \\ . & . & . & . \end{bmatrix}$, the value of PROJECT (A, MASK = L, FIELD = 0) is the scalar 8.

Case (ii): Using the arrays of case (i), the value of PROJECT (A, L, [0, 0, 0], DIM = 2) is the array [0, 8, 0], and the value of PROJECT (A, L, [0, 0, 0, 0], DIM = 1) is the array [0, 0, 8, 0].

The first nonzero number in each column of the table TABLE = $\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 3 & 5 & 0 \\ 1 & 4 & 6 & 0 \end{bmatrix}$ is located by the mask M = $\begin{bmatrix} . & . & . & . \\ . & T & . & . \\ . & . & T & . \\ T & . & . & . \end{bmatrix}$. A vector which contains those nonzero numbers can be extracted from TABLE by the PROJECT function. Thus, the value of PROJECT (TABLE, M, [−1, −1, −1, −1], DIM = 1) is that vector, namely [1, 2, 5, −1]. Note that M itself is the value of FIRSTLOC (TABLE.NE.0, DIM = 1).

F.2.4.3 REPLICATE (ARRAY, DIM, NCOPIES).

Description. Replicates an array by increasing a dimension.

Kind. Transformational function.

Arguments.

ARRAY	may be of any type. It must not be scalar.
DIM	must be scalar and of type integer with value in the range $1 \le \text{DIM} \le n$, where n is the rank of ARRAY.
NCOPIES	must be scalar and of type integer.

Result Type, Type Parameters, and Shape. The result is an array of the same type, type parameters, and rank as ARRAY and has shape [E (1:DIM-1), MAX (NCOPIES, 0) * E (DIM), E (DIM+1:*n*)], where the shape of ARRAY is E (1:*n*).

Result Value. Each element of the result has value equal to that of the corresponding element of ARRAY obtained by subtracting from subscript DIM sufficient integral multiples of E (DIM) to bring it into the range [1:E (DIM)].

Example. If A is the array $\begin{bmatrix} 2 & 3 \\ 3 & 4 \end{bmatrix}$, REPLICATE (A, DIM=2, NCOPIES=3) is $\begin{bmatrix} 2 & 3 & 2 & 3 & 2 & 3 \\ 3 & 4 & 3 & 4 & 3 & 4 \end{bmatrix}$.

F.2.4.4 RANK (SOURCE).

Description. Returns the rank of an array or a scalar.

Kind. Inquiry function.

Argument. SOURCE may be of any type.

Result Type and Shape. Integer scalar.

Result Value. The result has value zero if SOURCE is scalar and otherwise has value equal to the rank of SOURCE.

Example. RANK ([1:N]) has the value one.

F.2.4.5 FIRSTLOC (MASK, DIM).

Optional Argument. DIM

Description. Locate the leading edges of the set of true elements of a logical or bit mask.

Kind. Transformational function.

Arguments.

MASK	must be of type logical or bit. It must not be scalar. Its shape must be defined.
DIM (optional)	must be scalar and of type integer with value in the range $1 \le \text{DIM} \le n$, where n is the rank of MASK.

Result Type and Shape. The result is an array of the same shape as MASK and of the type of MASK.

Result Value.

Case (i): The result of FIRSTLOC (MASK) has at most one true element. If MASK is all false, the result is all false. If MASK contains one or more true elements, the result has a single true element and it is in the position corresponding to the first true element (in array element order) in MASK.

Case (ii): The result of FIRSTLOC (MASK, DIM) is defined by applying FIRSTLOC to each of the one-dimensional array sections of MASK that lie parallel to dimension DIM. Thus, section (s_1, s_2, ..., $s_{\text{DIM}-1}$, :, $s_{\text{DIM}+1}$, ..., s_n) of the

result has value equal to FIRSTLOC (MASK (s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n)).

Examples. If MASK is $\begin{bmatrix} . & . & T & . \\ . & T & T & . \\ . & T & . & T \\ . & . & . & . \end{bmatrix}$, where "T" represents .TRUE. and "." represents .FALSE., then

Case (i): FIRSTLOC (MASK) is $\begin{bmatrix} . & . & . & . \\ . & T & . & . \\ . & . & . & . \\ . & . & . & . \end{bmatrix}$ and

Case (ii): FIRSTLOC (MASK, DIM = 1) is the "top-edge" $\begin{bmatrix} . & . & T & . \\ . & T & . & . \\ . & . & . & T \\ . & . & . & . \end{bmatrix}$.

F.2.4.6 LASTLOC (MASK, DIM).

Optional Argument. DIM

Description. Locate the trailing edges of the set of true elements of a logical or bit mask.

Kind. Transformational function.

Arguments.

MASK must be of type logical or bit. It must not be scalar.

DIM (optional) must be scalar and of type integer with value in the range $1 \le DIM \le n$, where n is the rank of MASK.

Result Type and Shape. The result is an array of the same shape as MASK and of the type of MASK.

Result Value.

Case (i): The result of LASTLOC (MASK) has at most one true element. If MASK is all false, the result is all false. If MASK contains one or more true elements, the result has a single true element and it is in the position corresponding to the last true element (in array element order) in MASK.

Case (ii): The result of LASTLOC (MASK, DIM) is defined by applying LASTLOC to each of the one-dimensional array sections of MASK that lie parallel to dimension DIM. Thus, section (s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n) of the result has value equal to LASTLOC (MASK (s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n)).

Examples. If MASK is $\begin{bmatrix} . & . & T & . \\ . & T & T & . \\ . & T & . & T \\ . & . & . & . \end{bmatrix}$, where "T" represents .TRUE. and "." represents .FALSE., then

Case (i): LASTLOC (MASK) is $\begin{bmatrix} . & . & . & . \\ . & . & . & . \\ . & . & . & T \\ . & . & . & . \end{bmatrix}$ and

Case (ii): LASTLOC (MASK, DIM = 2) is $\begin{bmatrix} . & . & T & . \\ . & . & T & . \\ . & . & . & T \\ . & . & . & . \end{bmatrix}$.

F.3 Procedure Extensions.

F.3.1 Nesting of Internal Procedures. An internal procedure may host other internal procedures.

F.3.2 Internal Procedure Name as an Actual Argument. If a dummy argument is a dummy procedure, the associated actual argument must be the name of an external, internal, dummy, or intrinsic procedure.

If the internal procedure is supplied as an actual argument from an instance of the subprogram or a procedure having access to the entities of that instance, the instance of that internal procedure created by invoking the associated dummy procedure also has access by use or host association to the entities of that instance of the host subprogram.

All other entities, including saved data objects, are common to all instances of the subprogram. For example, the value of a saved data object appearing in one instance may have been defined in a previous instance or by a DATA attribute or DATA statement.

F.4 Condition Handling. This exception handling extension provides a structured way of dealing with relatively rare, synchronous events, such as errors in input data or instability of an algorithm near a critical point.

F.4.1 Definitions.

F.4.1.1 Condition. A **condition** is a named exceptional event or set of circumstances for which it is inappropriate to continue the normal execution sequence. Conditions may be user-defined or intrinsic to the processor. A processor must be able to detect the following intrinsic conditions:

(1) NUMERIC_ERROR. This condition occurs when the processor is unable to produce an acceptable result for an intrinsic numeric operation, either because the result is mathematically undefined or because the processor has no adequate representation for the result.

(2) BOUND_ERROR. This occurs when an array subscript, array section subscript, substring range expression, or effective range violates its bounds. This does not include violations of the requirements derived from the size of an assumed-size array.

(3) IO_ERROR. This condition occurs when an input/output error (9.4.3) is encountered in an input/output statement containing no IOSTAT= or ERR= specifier. If this condition is enabled, it may be handled as described below instead of causing immediate termination of the executable program.

(4) END_OF_FILE. This condition occurs when an end-of-file condition (9.4.3) is encountered in an input statement containing no IOSTAT= or END= specifier. If this condition is enabled, it may be handled as described below instead of causing immediate termination of the executable program.

(5) ALLOCATION_ERROR. This condition occurs when the processor is unable to perform an allocation requested by an ALLOCATE statement (6.2.2)

A processor may define additional intrinsic conditions.

Conditions may be passed as actual arguments and received as dummy arguments, known as dummy conditions.

F.4.1.2 Enabling. In order for an intrinsic condition to be detected automatically by the processor, it must be enabled. User-defined conditions may be enabled, though they need not be because they can be detected only by the user program itself. Dummy conditions must not be enabled.

F.4.1.3 Signaling. A condition may be signaled when the associated event or circumstance is detected. Conditions may be signaled explicitly by the execution of a SIGNAL statement (F.4.3.2) or, in the case of intrinsic conditions, implicitly by the processor.

F.4.1.4 Handler. Signaling a condition causes a transfer of control to a sequence of statements called a **condition handler**.

F.4.2 Specification Statements. The exception handling facility adds one new specification statement (CONDITION) as well as modifying an existing specification statement (INTRINSIC).

F.4.2.1 CONDITION Statement. A **CONDITION statement** is used to declare a user-defined or dummy condition. Rule R221 must be extended.

RF30 *specification-stmt* **is** *access-stmt*
or *condition-stmt*
or *data-stmt*
or *exponent-letter-stmt*
or *external-stmt*
or *intent-stmt*
or *intrinsic-stmt*
or *namelist-stmt*
or *optional-stmt*
or *range-stmt*
or *save-stmt*
or *common-stmt*
or *dimension-stmt*
or *equivalence-stmt*

RF31 *condition-stmt* **is** CONDITION [[, *condition-attr-spec*]... ::] *condition-name-list*

RF32 *condition-attr-spec* **is** OPTIONAL
or ENABLES (*condition-name-list*)
or HANDLES (*condition-name-list*)

Constraint: A *condition-name* must not be declared in more than one *condition-stmt* or *intrinsic-stmt* (F.4.2.2) in a scoping unit.

Constraint: There must not be more than one OPTIONAL attribute, one ENABLES attribute, and one HANDLES attribute in a *condition-stmt.*

Constraint: A dummy condition must not appear in either an ENABLES or a HANDLES attribute specification, nor may a dummy condition be declared in a *condition-stmt* that contains either of these attributes.

Constraint: The OPTIONAL attribute may appear only on a *condition-stmt* declaring a dummy condition.

Each name in a CONDITION statement (other than those in an ENABLES or HANDLES attribute specification) is declared to be a nonintrinsic condition. If the name also appears as a dummy argument in the current scope, it is a dummy condition; otherwise, it is a user-defined condition. Each condition name in an ENABLES or HANDLES attribute specification must be declared previously in a CONDITION statement or INTRINSIC statement (F.4.2.2).

Each request to enable one of the declared conditions is also a request to enable the conditions listed in the ENABLES attribute specification, if any. Each handler for one of the declared conditions is also a handler for the conditions in the ENABLES attribute specification, if any, except for those explicitly handled by other handlers in the same ENABLE construct (F.4.3.1).

Each handler for one of the declared conditions is also a handler for the conditions in the HANDLES attribute specification, if any, except for those explicitly handled by other handlers in the same ENABLE construct.

F.4.2.2 INTRINSIC Statement. An INTRINSIC statement also may be used to specify a name as representing an intrinsic condition (F.4.1.1). Rule R1206 must be extended.

R1206 *intrinsic-stmt* **is** INTRINSIC *intrinsic-name-list*

RF33 *intrinsic-name* **is** *intrinsic-procedure-name*
or *intrinsic-condition-name*

Each *intrinsic-name* must correspond to either an intrinsic procedure or an intrinsic condition supported by the processor.

If an intrinsic condition name is used as an actual argument to a nonintrinsic procedure, it must be declared in an INTRINSIC statement.

F.4.2.3 Implicit Declaration of Condition Names. If every occurrence of a name in a scoping unit is in an ENABLE or HANDLE statement (F.4.3.1), a SIGNAL statement (F.4.3.2), as the CONDITION argument to the ENABLED or HANDLED intrinsic functions (F.4.8.1, F.4.8.2), or as a dummy argument, the name is declared implicitly to be a condition name. If the name does not match any of the processor-supported intrinsic condition names, it identifies a user-defined condition; otherwise, it identifies the matching intrinsic condition.

F.4.2.4 Scope and Association of Condition Names. User-defined and intrinsic conditions are two separate classes of global entities of an executable program. User-defined conditions belong to the same class as program units, common blocks, and external procedures (14.1.1). A name that identifies a user-defined condition must not be used to identify any other global entities in this class. Intrinsic conditions belong to a second class of global entities. Within a scoping unit, a name that identifies an intrinsic condition must not be used to identify any other global entities; however, in a different scoping unit it may be used to identify a global entity of the first class.

Dummy conditions are local entities of the current scoping unit and of the same class as dummy procedures (14.1.2).

Conditions may be passed as actual arguments as described in 12.4.1. Rule R1211 must be extended.

RF34 *actual-arg* **is** *expr*
or *variable*
or *procedure-name*
or *condition-name*
or alt-return-spec

If a dummy argument is a dummy condition, the associated actual argument, if any, must be a condition. If the dummy condition has the OPTIONAL attribute and if no corresponding actual argument is supplied when the procedure is invoked, the dummy condition must not be signaled, nor supplied as the CONDITION argument to the intrinsic functions ENABLED or HANDLED. It may be supplied as an actual argument corresponding to an optional dummy condition. Then the optional dummy condition also is considered not to be associated with an actual argument. The PRESENT inquiry function may be used to determine if the dummy condition is associated with an actual condition.

F.4.3 Executable Constructs. The exception handling facility adds one new block construct (ENABLE) and a new action statement (SIGNAL).

F.4.3.1 ENABLE Construct. The ENABLE construct is used to enable the automatic detection of intrinsic conditions, supply handlers for conditions, and delimit a block that may be affected by the signaling of a condition. Rule R222 must be extended to include the ENABLE construct.

RF35 *executable-construct* **is** *action-stmt*
or *case-construct*
or *do-construct*
or *enable-construct*
or *if-construct*
or *where-construct*

RF36 *enable-construct* **is** *enable-stmt*
block
[*handle-stmt*
block]...
end-enable-stmt

RF37 *enable-stmt* **is** [*enable-construct-name* :] ■
■ ENABLE [(*condition-name-list*)]

RF38 *handle-stmt* **is** HANDLE (*condition-name-list*)
or HANDLE DEFAULT

RF39 *end-enable-stmt* **is** END ENABLE [*enable-construct-name*]

Constraint: A *condition-name* must not appear more than once in an *enable-stmt*.

Constraint: A *condition-name* appearing in an *enable-stmt* or *handle-stmt* must not be a dummy condition.

Constraint: HANDLE DEFAULT may appear at most once in an *enable-construct*.

Constraint: If an *enable-construct-name* is present, the same name must be specified on both the *enable*stmt and the corresponding *end-enable-stmt*.

The block immediately following the ENABLE statement is the **ENABLE block**. Each block following a HANDLE statement is called a **HANDLE block**.

A condition name must not appear explicitly in more than one HANDLE statement of an ENABLE construct. If a condition name does not appear explicitly in any HANDLE statements of an ENABLE construct, it must not be implied directly or indirectly, via HANDLES or ENABLES attributes (F.4.2.1) in the same scoping unit, by CONDITION names listed on more than one HANDLE statement of the construct.

Both the ENABLE statement and the END ENABLE statement are branch target statements (8.2); however, it is permissible to branch to an END ENABLE statement only from within its ENABLE construct.

F.4.3.2 SIGNAL Statement. Any condition, including intrinsic and dummy conditions, may be signaled explicitly by a **SIGNAL statement**. Rule R223 must be extended to include the SIGNAL statement.

RF40 *action-stmt* **is** *allocate-stmt*
or *assignment-stmt*
or *backspace-stmt*
or *call-stmt*
or *close-stmt*
or *computed-goto-stmt*
or *continue-stmt*
or *cycle-stmt*
or *deallocate-stmt*

		or *endfile-stmt*
		or *exit-stmt*
		or *forall-stmt*
		or *goto-stmt*
		or *identify-stmt*
		or *if-stmt*
		or *inquire-stmt*
		or *open-stmt*
		or *print-stmt*
		or *read-stmt*
		or *return-stmt*
		or *rewind-stmt*
		or *set-range-stmt*
		or *signal-stmt*
		or *stop-stmt*
		or *where-stmt*
		or *write-stmt*
		or arithmetic-if-stmt
		or assign-stmt
		or assigned-goto-stmt
		or pause-stmt
RF41	*signal-stmt*	**is** SIGNAL (*condition-name*)
		or SIGNAL (*)

Constraint: SIGNAL (*) is permitted only in a HANDLE block.

F.4.4 Condition Enabling. All conditions that are enabled for the ENABLE statement itself remain enabled throughout the ENABLE construct. Any other conditions in the condition name list, if any, of the ENABLE statement, including those implied, either directly or indirectly, by any ENABLES attributes (F.4.2.1) in the current scoping unit, are enabled only within the ENABLE block. Enabling a condition in one procedure does not enable that condition in any procedure invoked from within the ENABLE block.

F.4.5 Condition Signaling. A condition is **signaled indeterminately** if it is detected during expression evaluation or assignment. An indeterminately signaled condition affects entities in the innermost ENABLE block or scoping unit that contains the operation causing the signal. If circumstances are such that two independent operations could each signal a condition indeterminately in the same ENABLE block, the condition that serves as the basis for transfer of control is processor dependent.

A condition is **signaled determinately** if it is detected in any other way. A determinately signaled condition can affect only entities in the statement in which the condition is detected.

The intrinsic conditions, if they are enabled, are signaled implicitly by the processor whenever the events they represent are detected.

Execution of a SIGNAL statement determinately signals the condition indicated by the condition name that appears in the statement. If the SIGNAL statement appears in a HANDLE block and the condition name is specified by *, the condition signaled is the condition that caused the transfer to the block. Signaling a dummy condition is equivalent to signaling the corresponding actual argument in the current scoping unit. A condition need not be enabled to be signaled explicitly.

F.4.6 Execution of an ENABLE Construct. Execution of an ENABLE construct begins with the first executable construct of the ENABLE block, and continues to the end of the block unless a condition is signaled. If no condition is signaled anywhere within the ENABLE block, the execution of the entire construct is complete when the execution of the ENABLE block is complete.

F.4.6.1 Condition Handling. If a condition is signaled in an ENABLE block and the ENABLE construct either:

(1) Contains a HANDLE statement that explicitly lists the condition, or

(2) Contains no HANDLE statement that explicitly list the condition, but does contain a HANDLE statement that implies the condition, either directly or indirectly, via ENABLES or HANDLES attributes in the same program unit (F.4.2.1), or

(3) Contains no HANDLE statement that lists or implies the condition, but does contain a HANDLE DEFAULT statement

the associated HANDLE block is called the **handler** for that condition and the ENABLE construct is said to **supply the handler.** An ENABLE construct never supplies a handler for a condition detected in one of its HANDLE blocks. The block following the HANDLE DEFAULT statement is called the **default handler** for that ENABLE construct. It handles all conditions not otherwise handled in that ENABLE construct.

When a condition is signaled, control is transferred to the HANDLE block supplied by the innermost ENABLE construct that supplies a handler for that condition. Execution of the HANDLE block completes the execution of the ENABLE construct.

F.4.6.2 Condition Propagation. If a condition is signaled, but no handler is supplied in the current scoping unit, the condition is **propagated**. A condition must not be propagated from a main program. A condition, either intrinsic, user-defined, or dummy, is propagated from a function or subroutine by signaling it in the invoking procedure, regardless of whether it was enabled in that procedure. If the current procedure was invoked during expression evaluation or assignment, the condition is signaled indeterminately in the invoking procedure, either in the innermost ENABLE block or in the entire scoping unit. Otherwise, it is signaled determinately in the statement invoking the current procedure.

F.4.7 Effects of Signaling Events on Definition. The signaling of a condition also may cause entities to become undefined (14.8.6). When a condition is signaled determinately in a statement, the entities affected are those whose definition status could have been affected by the statement had no condition been signaled, with one exception: if the statement is a READ statement with a VALUES= specifier and if the signaled condition is either IO__ERROR or END__OF__FILE, the specified variable and, possibly, some or all of the variables in the input/output list become defined as described in 9.4.3.

When a condition is signaled indeterminately in an ENABLE block, the entities affected are those whose definition status has been affected or could have been affected by statements in the block had no condition been signaled.

When a condition is signaled indeterminately outside any ENABLE block, the entities affected are those whose definition status has been affected or could have been affected by statements anywhere in the scoping unit had no condition been signaled.

F.4.7.1 Examples of ENABLE Constructs. Example 1:

```
IO_CHECK:  ENABLE (IO_ERROR, END_OF_FILE)
   . . .
   READ (*, '(I5)') I
   . . .
   READ (*, '(I5)', END = 90) J
   . . .
```

```
      GO TO 100
90    CONTINUE
      J = 0
      GO TO 100
      HANDLE (END_OF_FILE)
         WRITE (*, *) 'UNEXPECTED END-OF-FILE'
         STOP
      HANDLE (IO_ERROR)
         WRITE (8, *) 'I/O ERROR'
         STOP
      END ENABLE IO_CHECK
100   CONTINUE
```

In this example, if an input/output error occurs in either of the READ statements or if an end-of-file is encountered in the first READ statement, the appropriate condition will be signaled determinately (thus affecting only the value of the variable in the input/output list), and a handler will receive control, print a message, and terminate the program. However, if an end-of-file is encountered in the second READ statement, no condition will be signaled and control will be transferred to the statement indicated in the END= specifier.

Example 2:

```
PROGRAM EXAMPLE
ENABLE (SINGULARITY_ERROR)
   ENABLE
      . . . ! FIRST TRY THE "FAST" ALGORITHM:
      CALL FAST_INV (AMATRIX, VMATRIX, SDET, ESIZE (AMATRIX, 1))
   HANDLE (SINGULARITY_ERROR)
      . . . ! "FAST" ALGORITHM FAILED; TRY "SLOW" ONE:
      CALL SLOW_INV (AMATRIX, VMATRIX, SDET, ESIZE (AMATRIX, 1))
   END ENABLE
HANDLE (SINGULARITY_ERROR)
   WRITE (*, *) 'CANNOT INVERT MATRIX'
   STOP
END ENABLE
STOP

CONTAINS

! HERE'S FAST_INV:

SUBROUTINE FAST_INV (AMAT, VMAT, DET, NMAT)
REAL AMAT (NMAT, NMAT), VMAT (NMAT, NMAT)
VMAT = 0
ENABLE (NUMERIC_ERROR)
   . . .
   ENABLE
      DET = DETERMINANT (AMAT, NMAT)
   END ENABLE
   . . .
HANDLE (SINGULARITY_ERROR, NUMERIC_ERROR)
   DET = 0
   SIGNAL (SINGULARITY_ERROR)
END ENABLE
END SUBROUTINE FAST_INV

! ASSUME SLOW_INV IS AN EXTERNAL ROUTINE

! AND HERE'S DETERMINANT:
```

```
REAL FUNCTION DETERMINANT (X, N)
INTEGER, INTENT (IN) :: N
REAL, INTENT (IN)    :: X (N, N)
ENABLE (NUMERIC_ERROR)
   . . .
   IF (DIAG == 0) SIGNAL (SINGULARITY_ERROR)
   . . .
   DETERMINANT = . . .
HANDLE (SINGULARITY_ERROR, NUMERIC_ERROR)
   . . . ! CLEANUP
   SIGNAL (*)
END ENABLE
END FUNCTION DETERMINANT
END PROGRAM EXAMPLE
```

Assume NUMERIC_ERROR is signaled implicitly somewhere inside DETERMINANT. The handler does any necessary cleanup, then simply resignals NUMERIC_ERROR. Since there is no further handler in DETERMINANT, the condition is propagated. In FAST_INV, NUMERIC_ERROR is signaled indeterminately because DETERMINANT was invoked during expression evaluation; however, the invocation of DETERMINANT is bracketed by ENABLE and END ENABLE, and the arguments are INTENT (IN), so only DET becomes defined. The handler sets DET to zero and remaps the condition by signaling SINGULARITY_ERROR, which is then propagated because there is no further handler in FAST_INV. In the host, SINGULARITY_ERROR is signaled determinately in the call to FAST_INV, so control is passed to the first handler. Here an external subroutine with a better but slower algorithm is called. If this routine also signals SINGULARITY_ERROR, control is passed to the second handler, which gives up and terminates the program.

F.4.8 Condition Status Inquiry Functions. The inquiry functions ENABLED and HANDLED permit inquiries to be made about whether a condition has been enabled or would be handled.

F.4.8.1 ENABLED (CONDITION, LEVEL).

Optional Argument. LEVEL

Description. Determine whether a condition is enabled.

Kind. Inquiry function.

Arguments.

CONDITION	must be a condition name.
LEVEL (optional)	must be scalar and of type integer. Its value must not be negative. If omitted, the result is determined as though LEVEL were present with value 1.

Result Type and Shape. Logical scalar.

Result Value. The result is defined recursively, as follows:

Case (i): If the condition specified by CONDITION is enabled, the result is .TRUE.

Case (ii): If case (i) does not apply and either LEVEL is zero or the current scoping unit is that of a main program, the result is .FALSE.

Case (iii): If neither of the first two cases hold, the result is that of ENABLED (CONDITION, LEVEL – 1) evaluated at the point of reference to the current procedure.

F.4.8.2 HANDLED (CONDITION, LEVEL).

Optional Argument. LEVEL

Description. Determine whether a condition would be handled.

Kind. Inquiry function.

Arguments.

CONDITION	must be a condition name.
LEVEL (optional)	must be scalar and of type integer. Its value must not be negative. If omitted, the result is determined as though LEVEL were present with value HUGE (0).

Result Type and Shape. Logical scalar.

Result Value. The result is defined recursively as follow:

Case (i): If a handler is supplied for an occurrence of the condition specified by CONDITION, the result is .TRUE.

Case (ii): If no such handler is supplied and either LEVEL is zero or the current scoping unit is that of a main program, the result is .FALSE.

Case (iii): If neither of the first two cases hold, the result is that of HANDLED (CONDITION, LEVEL – 1) evaluated at the point of reference to the current procedure.

F.4.9 Notes on Exception Handling. Intrinsic conditions that correspond to violations of language or processor restrictions also may be signaled by the processor even if not enabled. However, programs that rely on such behavior are not standard conforming. Moreover, the result returned by the ENABLED intrinsic inquiry function must not depend on the presence of absence of such processor extensions.

F.5 Significant Blanks in Free Source Form. In free form, blank characters are significant and must not appear within lexical tokens and must be used to separate a name adjacent to a keyword. A sequence of blank characters outside of a character context is equivalent to a single blank character.

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

APPENDIX G INDEX

(This appendix is not part of American National Standard X3.9-198x, but is included for information only.)

APPENDIX H. GLOSSARY OF TECHNICAL TERMS

(This appendix is not part of American National Standard X3.9-198x, but is included for information only.)

The following is a list of the principal technical terms used in the standard and their definitions.

access (11.3.2). The USE statement provides the means by which a scoping unit accesses entities in a module subprogram. Such entities may be explicitly or implicitly accessible.

action statement. A single statement specifying a computational action. See BNF R223.

alias (6.2.6). A data object that has a type, type parameters, and a rank. It may not be referenced or defined unless it is alias associated with a data object that may be referenced or defined. If it is an array, it does not have a shape unless it is alias associated.

alias association (6.2.6). Following a valid execution of an IDENTIFY statement, an alias is alias associated with a nonalias object.

allocatable array (5.1.2.4.3). A named array that has a type, type parameters, and a rank, but only when it has space allocated for it does it have a shape and may it be referenced or defined.

argument keyword (2.5.2). A dummy argument name. It may used in a procedure reference ahead of the equals symbol (see BNF R1209) provided the procedure has an explicit interface.

array (2.4.7). A data object composed of scalar data of the same type and type parameters that are arranged in a rectangular pattern. It may be a named array, an array section, a structure component, an array-valued function result, or an array-valued expression. Its rank is at least one.

array element (6.2). The scalar data that make up an array are known as the array elements (see BNF R614).

array section (6.2.4.3). An array subobject designated by the name of an array followed by a section subscript list, optionally followed by a substring range (see BNF R615).

association (2.5.6). An association exists if an entity may be identified by different names in the same scoping unit or by the same name or different names in different scoping units.

attributes (5). Properties of a data object that may be specified in a type declaration statement, namely type, type parameters, rank, shape, whether variable or constant, initial value, accessibility (PUBLIC or PRIVATE), intent (IN, OUT, or INOUT), whether allocatable, whether an alias, whether optional, whether to be saved, and whether ranged. See BNF R501.

belong (8.1.4.1). If an EXIT or CYCLE statement contains a construct name, it belongs to the DO construct using that name. Otherwise, it belongs to the innermost DO construct in which it appears.

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.

component (4.4). A derived type is defined as a set of components. See BNF R416.

conformable (2.4.7). Two arrays are said to be conformable if they have the same shape. A scalar is conformable with any array.

conformance (1.4). An executable program conforms to this standard if it uses only those forms and relationships described herein and if the executable program has an interpretation according to this standard. A program unit conforms to this standard if it can be included in an executable program in a manner that allows the executable program to be standard conforming. A processor conforms to the standard if it executes standard-conforming programs in a manner that fulfills the interpretations prescribed in the standard.

connected (9.3.2). If a unit is connected, it refers to a file and the file is connected to the unit.

constant (2.4.4). A named constant or a literal constant.

data entity (2.4.3, 4.2). An entity that has or may have a data value. It may be a constant, a variable, an expression value, or a function result.

data object (2.4.3.1). A datum or a set of data of the same type and type parameters that may be referenced as a whole. It may be named or it may be a subobject.

data type (2.4.1). A data type is 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. A type may be parameterized, in which case the set of data values depends on the values of the parameters.

deferred-shape array (5.1.2.4.3, 6.2.6). An allocatable array or an alias array.

definable (2.5.4). A variable is definable if its value may be changed as a whole. An allocatable array that has not been allocated is an example of a data object that is not definable. Aliases for which an IDENTIFY statement has not been executed are examples of data objects that are not definable. An example of a subobject that is not definable is C (I) when C is a constant array and I is an integer variable.

defined assignment statement (7.5.1.3). An assignment statement that is not an intrinsic assignment statement and is defined by a subroutine subprogram whose interface is explicit.

defined operation (7.1.3). An operation that is not an intrinsic operation and is defined by a function subprogram whose interface is explicit.

definition of a procedure or type (2.5.4). A procedure is defined by a subprogram and a derived type is defined by a sequence of statements commencing with a TYPE statement and terminating with an END TYPE statement. See BNF R416.

definition of a variable (2.5.4). When a variable is given a valid value during program execution, it is said to become defined. Under certain other circumstances it ceases to have a valid value and is said to become undefined.

deleted features (1.6). Features in ANSI X3.9-1978 that are considered to have been redundant and are largely unused. These features are not included in this revision of Fortran.

deprecated features (1.6). Features in ANSI X3.9-1978 that are considered to have become redundant by the inclusion of certain new features in this standard. They may become obsolescent as the new features become widely used.

derived type (2.4.1.2). A type whose data have components of intrinsic types and other derived types.

designator. See subobject designator.

dummy argument (12.5.2.2, 12.5.2.3, 12.5.4). An entity whose name appears in a dummy argument list in a FUNCTION or SUBROUTINE statement or statement function.

elemental (12.4.3, 12.4.5). An operation, function, or assignment that is applied independently to the elements of an array or corresponding elements of a set of conformable arrays and scalars.

entity (2.1). The term entity is used for any of the following: a program unit, a procedure, an operator, an interface block, a common block, an i/o unit, a statement function, a type, a named variable, an expression, a component of a type, a named constant, a statement label, a construct, an exponent letter, a namelist group, or a range list.

executable construct (2.1). A CASE, DO, ENABLE, IF, or WHERE construct or an action statement. See BNF R219.

executable program (2.2.2). A set of program units that includes exactly one main program.

executable statement (2.3.1). An instruction to perform or control one or more computational actions. The executable statements are all those that make up the syntactic class of executable-construct. See BNF R222.

expression (7.1). An expression is formed from operands, operators, and parentheses (see BNF R712). It may be a variable, a constant, a function reference, or may represent a computation.

extent (2.4.7). The size of one dimension of an array.

external procedure (2.2.4.1). A procedure that is defined by an external subprogram or by means other than Fortran.

external subprogram (R204). A subprogram that is not contained in a main program, module, or another subprogram.

function (2.2.4). A procedure that is invoked in an expression.

function subprogram (2.1). A subprogram whose first statement is a FUNCTION statement. See BNF R204, R205, R218, R220.

global entity (14). An entity identified by a lexical token whose scope is an executable program. It may be a program unit, a common block, or an external procedure.

host (2.2.4.3). A main program or subprogram that contains an internal procedure is called the host of the internal procedure. A module that contains a module procedure is called the host of the module procedure.

instance of a subprogram (12.5.2.4). When a function or subroutine defined by a subprogram is invoked, an instance of that subprogram is created.

interface of a procedure (12.3). See procedure interface.

internal procedure (2.2.4.3). A procedure that is defined by an internal subprogram (R218).

intrinsic (2.5.7). Intrinsic types, operators, and procedures are defined in the standard and may be used in any scoping unit without further definition or specification.

invoke (2.2.4). A subroutine is invoked by a CALL statement or by a defined assignment statement. A function is invoked by a reference to it by name or operator during the evaluation of an expression.

keyword (2.5.2). Statement keyword or argument keyword.

length (4.3.2.1). The length of a character string is its number of characters.

literal constant (2.4.4). A lexical token that directly represents a scalar value of intrinsic type. See BNF R306.

local entity (14). An entity identified by a lexical token whose scope is a scoping unit.

main program (2.2.3, 11.1). A program unit that is not a module, subprogram, or block data program unit. See BNF R203.

module (2.2.5, 11.3). A program unit that contains or accesses definitions to be accessed by other program units.

module procedure (2.2.4.2). A procedure that is defined by a module subprogram (R220).

name (3.2.2). A string consisting of a letter followed by up to 30 alphanumeric characters (letters, digits, and underscores).

named constant (2.4.4). A named data object whose value must not change during execution of an executable program.

object (2.4.3.1). Data object.

obsolescent features (1.6). Features in ANSI X3.9-1978 that are considered to have been redundant but that are still in frequent use.

operator (2.5.8). An operator specifies a particular computation involving one or two operands.

parent of an alias (6.2.6). The data object with which an alias appears in an IDENTIFY statement.

present (12.5.2.8). A dummy argument is present in an instance of a subprogram if there is a corresponding actual argument and the actual argument is a dummy argument that is present in the invoking procedure or is not a dummy argument of the invoking program unit.

procedure (2.2.4, 12.1.2). A computation that may be invoked during program execution. It may be a function or a subroutine. It may be 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 the procedure, the name of the procedure, the name of each dummy argument, the operator (if any) by which it may be referenced (functions only), and whether it may be referenced by an assignment statement (subroutines only).

processor (1.2). The combination of a computing system and the mechanism by which programs are transformed for use on that computing system.

program. See executable program and main program.

program unit (2.2). The fundamental component of a Fortran program. A sequence of statements and comment lines. It may be a main program, a module, an external subprogram, or a block data program unit.

rank (2.4.7). The number of dimensions of an array. It is zero for a scalar.

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, or the appearance of a procedure name, its operator symbol, or a defined assignment in a context requiring execution of the procedure at that point. Note that 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.

scalar (2.4.6). A single datum that is not array valued.

scoping unit (2.2.1). One of the following:

(1) A derived-type definition,

(2) A procedure interface block, excluding any procedure interface blocks contained within it, or

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

sequence array (12.4.1.4). An assumed-size array or an explicit-shape array without the RANGE attribute that is either a dummy array associated with a sequence array or is not a dummy argument.

shape (2.4.7). The shape of an array is the rank-one array whose elements are the extents in each dimension.

size (2.4.7). The size of an array is the total number of elements.

standard module (1.7). A module standardized as a separate collateral standard.

statement (2.3). A program unit is a sequence of statements. The way it is divided into statements is explained in 3.3.

statement function (12.5.4). A procedure specified by a single statement that is similar in form to a scalar 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 entity (14). An entity identified by a lexical element whose scope is a single statement or part of a statement.

storage association (14.7.2.2). Two storage sequences are associated if a storage unit of one is the same as a storage unit of the other.

storage sequence (14.7.2.1). A sequence of storage units.

stride (6.2.4.4). The increment specified by a section subscript triplet (BNF R620, R621).

structure (4.2). An object of derived type.

subobject (2.4.3.2). Part of a data object. It may be an array element, an array section, a structure component, or a substring.

subobject designator (2.5.1). A subobject designator is a name, followed by one or more of component selectors, array section selectors, array element selectors, and substring selectors.

subprogram (2.1). An external subprogram (R204), an internal subprogram (R218), or a module subprogram (R220). Modules and block data program units are not subprograms.

subroutine (2.2.4). A procedure that is invoked by a CALL statement or by a defined assignment statement.

subroutine subprogram (2.1). A subprogram whose first statement is a SUBROUTINE statement. See BNF R204, R205, R218, R220.

subscript (6.2.4). An element of a named array or an array-valued structure component is selected by a list of subscripts. Note that in FORTRAN 77, the whole list was called the subscript.

substring (6.1.1). A contiguous portion of a scalar character string. Note that an array section can include a substring-range; the result is called a section and not a substring.

type (4). Data type.

type parameter (1.5.3, 2.4.1). A parameter of a parameterized data type.

type parameter values (1.5.3, 2.4.1). The values of the type parameters of a data entity of parameterized data type.

value attribute (5.1.2.1). Whether a data object is constant or variable and whether it has a defined initial value.

variable (2.4.5). A data object that is not a constant. It may be a named variable, an array element, an array section, a structure component, or a substring.

whole array (6.2.1). A named array.

The following is a list of the Appendix F extensions to the meanings of the above technical terms:

definable. Many-to-one vector-valued array sections are examples of data subobjects that are not definable.

entity. A condition is an entity.

host. In the case of nested internal procedures, the host is the program unit that immediately contains the internal procedure. For example, if A contains B contains C, A is the host of B and B is the host of C, but A is not the host of C.

intrinsic. There are also intrinsic conditions.

The following is a list of important additional technical terms used in Appendix F, together with their definitions:

condition (F.4.1.1). A named circumstance in which it is inappropriate to continue the normal execution sequence.

element array assignment statement (F.2.3). An assignment statement for arrays that specifies the array elements to be assigned in terms of array elements, array sections, and scalar logical and bit masks.

handler (F.4.1.4). A sequence of statements commencing with a HANDLE statement and ending with the statement before the next HANDLE or END ENABLE statement, whichever comes first. A handler is executed when a condition specified for it is signaled.

many-to-one vector subscript. A vector subscript that has two or more elements with the same value.

signal. A condition is signaled in a named circumstance when it is inappropriate to continue the normal execution sequence.

vector subscript. A section subscript that is a rank-one integer expression.