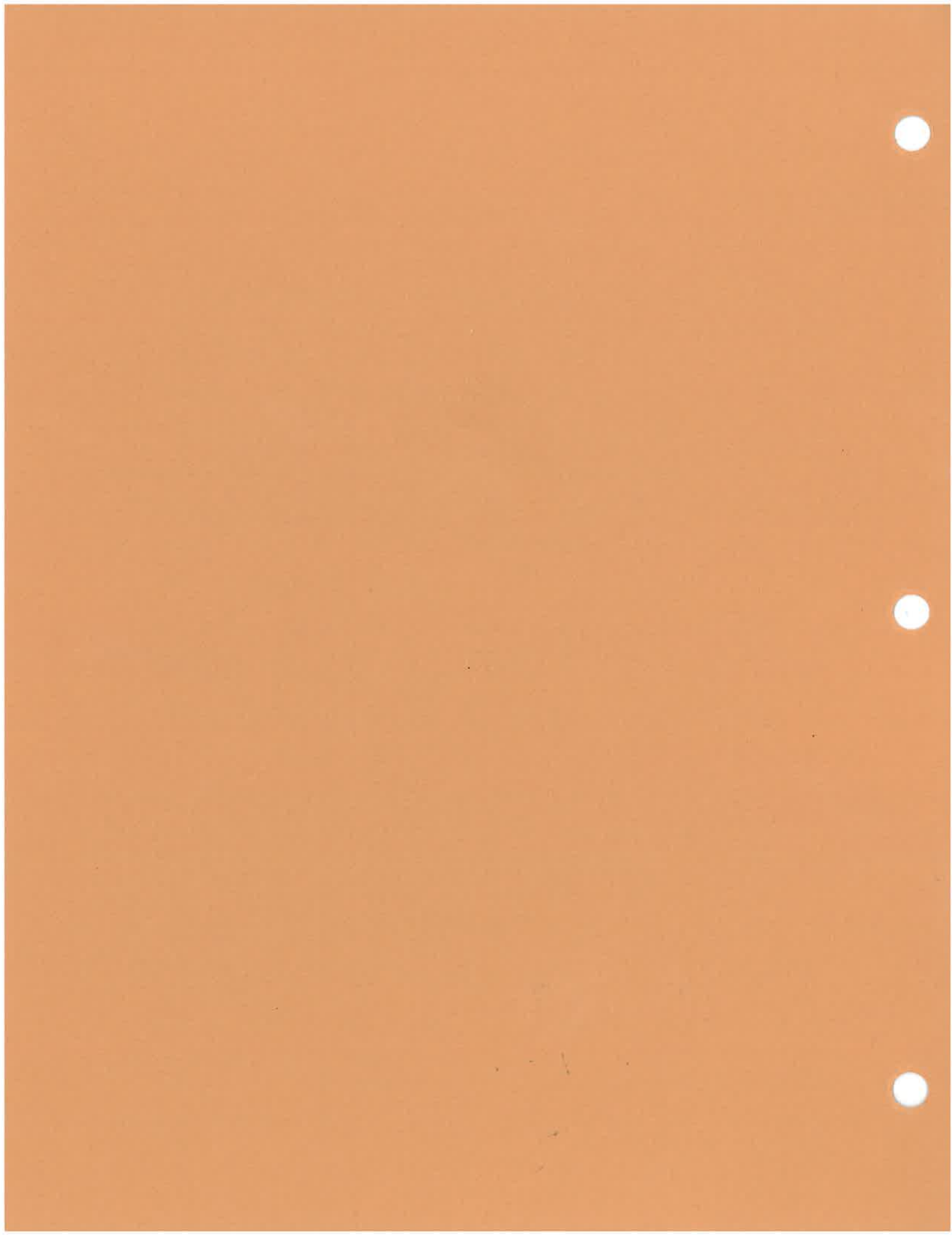


X3J3 / S8.115

June 1990

# Fortran 90

THIS IS AN INTERNAL WORKING DOCUMENT  
OF X3J3, THE FORTRAN TECHNICAL COMMITTEE  
OF THE AMERICAN NATIONAL STANDARDS INSTITUTE



**Draft International Standard**

# **F o r t r a n**

**Revision of ISO-1539**

**Submitted to ISO by X3J3,  
Document Editor for the JTC1/SC22/WG5  
Draft International Fortran Standard**



# FOREWORD

1 (This foreword is not part of this standard.)

2 **Standard Programming Language Fortran.** This standard specifies the form and estab-  
3 lishes the interpretation of programs expressed in the Fortran language (known informally as  
4 "Fortran 90"). It consists of the specification of the language Fortran. No subsets are specified in  
5 this standard. With limitations noted in 1.4.1, the syntax and semantics of the standard commonly  
6 known as "FORTRAN 77" are contained entirely within this standard. Therefore, any standard-  
7 conforming FORTRAN 77 program is standard conforming under this standard. New features can be  
8 compatibly incorporated into such programs, with any exceptions indicated in the text of this  
9 standard.

10 A standard-conforming Fortran processor is also a standard-conforming FORTRAN 77 processor.

11 Note that the name of this language, Fortran, differs from that in FORTRAN 77 in that only the first  
12 letter is capitalized. Both FORTRAN 77 and FORTRAN 66 used only capital letters in the official name  
13 of the language, but Fortran 90 does not continue this tradition.

## 14 OVERVIEW

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

- 16 (1) Array operations
- 17 (2) Improved facilities for numerical computation
- 18 (3) Parameterized intrinsic data types
- 19 (4) User-defined data types
- 20 (5) Facilities for modular data and procedure definitions
- 21 (6) Pointers
- 22 (7) The concept of language evolution

23 A number of other additions are also included in this standard, such as improved source form  
24 facilities, more control constructs, recursion, additional input/output facilities, and dynamically  
25 allocatable arrays.

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

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

39 **Numerical Computation.** Scientific computation is one of the principal application domains  
40 of Fortran, and a guiding objective for all of the technical work is to strengthen Fortran as a vehicle  
41 for implementing scientific software. Though nonnumeric computations are increasing dramati-  
42 cally in scientific applications, numeric computation remains dominant. Accordingly, the addi-  
43 tions include portable control over numeric precision specification, inquiry as to the characteristics  
44 of numeric representation, and improved control of the performance of numerical programs (for  
45 example, improved argument range reduction and scaling).



1 **Parameterized Character Data Type.** Optional facilities for multibyte character data for  
 2 languages with large character sets, such as those in China and Japan, are added by using a kind  
 3 parameter for the character data type. This facility allows additional character sets for special pur-  
 4 poses as well, such as characters for mathematics, chemistry, or music.

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

12 Procedure definitions may be used to define operations on intrinsic or derived types and nonin-  
 13 trinsic assignments for intrinsic and derived types.

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

26 **Pointers.** Pointers allow arrays to be sized dynamically and ranged, and structures to be linked  
 27 to create lists, trees, and graphs. An object of any intrinsic or derived type may be declared to  
 28 have the pointer attribute. Once such an object becomes associated with a target, it may appear  
 29 almost anywhere a nonpointer object with the same type, type parameters, and shape may appear.

30 **Language Evolution.** With the addition of new facilities, certain old features become redun-  
 31 dant and may eventually be phased out of the language as their usage declines. For example, the  
 32 numeric facilities alluded to above provide the functionality of double precision; with the new  
 33 array facilities, nonconformable argument association (such as associating an array element with a  
 34 dummy array) is unnecessary (and in fact is not useful as an array operation); and block data pro-  
 35 gram units are redundant and inferior to modules.

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

## 38 DOCUMENT ORGANIZATION

39 This document is organized in 14 sections, dealing with 7 conceptual areas. These 7 areas, and the  
 40 sections in which they are treated, are:

41	High/Low Level Concepts	Sections 2,3
42	Data Concepts	Sections 4,5,6
43	Computations	Sections 7,13
44	Execution Control	Section 8
45	Input/Output	Sections 9,10
46	Program Units	Sections 11,12
47	Scoping and Association Rules	Section 14





- 1 **High/Low Level Concepts.** Section 2 (Fortran Terms and Concepts) contains many of the high  
2 level concepts of Fortran. This includes the concept of an executable program and the relation-  
3 ships among its major parts. Also included are the syntax of program units, the rules for statement  
4 ordering, and the definitions of many of the fundamental terms used throughout the document.
- 5 Section 3 (Characters, Lexical Tokens, and Source Form) describes the low level elements of For-  
6 tran, such as the character set and the allowable forms for source programs. It also contains the  
7 rules for constructing literal constants and names for Fortran entities, and lists all of the Fortran  
8 operators.
- 9 **Data Concepts.** The array operations (arrays as data objects) and data structures provide a rich  
10 set of data concepts in Fortran. The main concepts are those of data type, data object, and the use  
11 of data objects, which are described in Sections 4, 5, and 6, respectively.
- 12 Section 4 (Intrinsic and Derived Data Types) describes the distinction between a data type and a  
13 data object, and then focuses on data type. It defines a data type as a set of data values, corre-  
14 sponding forms (constants) for representing these values, and operations on these values. The  
15 concept of an intrinsic data type is introduced, and the properties of Fortran's intrinsic types  
16 (INTEGER, REAL, COMPLEX, LOGICAL, and CHARACTER) are described. Note that only type  
17 concepts are described here, and not the declaration and properties of data objects.
- 18 Section 4 also introduces the concept of derived (user-defined) data types, which are compound  
19 types whose components ultimately resolve into intrinsic types. The details of defining a derived  
20 type are given (note that this has no counterpart with intrinsic types as intrinsic types are pre-  
21 defined and therefore need not—indeed cannot—be redefined by the programmer). As with  
22 intrinsic types, this section deals only with type properties, and not with the declaration of data  
23 objects of derived type.
- 24 Section 5 (Data Object Declarations and Specifications) describes in detail how named data objects  
25 are declared and given the desired properties (attributes). An important attribute (the only one  
26 required for each data object) is the object's data type, so the type declaration statement is the main  
27 feature of this section. The various attributes are described in detail, as well as the two ways that  
28 attributes may be specified (type declaration statements and attribute specification statements).  
29 Implicit typing and storage association (COMMON and EQUIVALENCE) are also described in  
30 this section, as well as data object value initialization.
- 31 Section 6 (Use of Data Objects) deals mainly with the concept of a variable, and describes the vari-  
32 ous forms that variables may take. Scalar variables include character strings and substrings, struc-  
33 tured (derived-type) objects, structure components, and array elements. Arrays are considered to  
34 be variables, as are array sections. Among the array facilities described here are array sections  
35 (subarrays), and array allocation and deallocation (user controlled dynamic arrays). The section  
36 concludes with a summary of the allowed appearances of array names.
- 37 **Computations.** Section 7 (Expressions and Assignment) describes how computations are  
38 expressed in Fortran. This includes the forms that expression operands (primaries) may take and  
39 the role of operators in these expressions. Operator precedence is rigorously defined in syntax  
40 rules and summarized in tabular form. This description includes the relationship of defined oper-  
41 ators (user-defined operators) to the intrinsic operators (+, \*, .AND., .OR., etc.). The rules for both  
42 expression evaluation and the interpretation (semantics) of intrinsic and defined operators are  
43 described in detail.
- 44 Section 7 also describes assignment of computational results to data objects, which has three prin-  
45 cipal forms: the conventional assignment statement, the pointer assignment statement, and the  
46 WHERE statement and construct. The WHERE statement and construct allow masked array  
47 assignment.
- 48 Section 13 (Intrinsic Procedures) describes more than one hundred intrinsic procedures that pro-  
49 vide a rich set of computational capabilities. In addition to the FORTRAN 77 intrinsic functions, this  
50 includes many array processing functions, a comprehensive set of numerical environmental  
51 inquiry functions, and a set of procedures for the manipulation of bits in nonnegative integer data.



- 1 **Execution Control.** Section 8 (Execution Control) describes the control constructs (IF, CASE,  
2 and DO), branching statements (various forms of GO TO), and other control statements (IF, arith-  
3 metic IF, CONTINUE, STOP, and PAUSE). These are as in FORTRAN 77 except for the addition of  
4 the CASE construct and the extension of the DO loop to include an END DO termination option,  
5 additional control clauses, and the addition of the EXIT and CYCLE statements.
- 6 **Input/Output.** Section 9 (Input/Output Statements) contains definitions for records, files, file  
7 connections (OPEN, CLOSE, and preconnected files), data transfer statements (READ, WRITE, and  
8 PRINT) that include processing of partial and variable length records, file positioning, and file  
9 inquiry (INQUIRE).
- 10 Section 10 (Input/Output Editing) describes input/output formatting. This includes the FORMAT  
11 statement and FMT= specifier, edit descriptors, list-directed I/O, and namelist I/O.
- 12 **Program Units.** Section 11 (Program Units) describes main programs, external subprograms,  
13 modules, and block data program units. Modules, along with the USE statement, are described as  
14 a mechanism for encapsulating data and procedure definitions that are to be used by (accessible to)  
15 other program units. Modules are described as vehicles for defining global derived-type defini-  
16 tions, global data object declarations, procedure libraries, and combinations thereof.
- 17 Section 12 (Procedures) contains a comprehensive treatment of procedure definition and invoca-  
18 tion, including that for user-defined functions and subroutines. The concepts of implicit and  
19 explicit procedure interfaces are explained, and situations requiring explicit procedure interfaces  
20 are identified. The rules governing actual and dummy arguments, and their association, are  
21 described.
- 22 Section 12 also describes the use of the OPERATOR option on interface blocks to allow function  
23 invocation in the form of infix and prefix operators as well as the traditional functional form. Simi-  
24 larly, the use of the ASSIGNMENT option on interface blocks is described as allowing an alternate  
25 syntax for certain subroutine calls. This section also contains descriptions of recursive procedures,  
26 the RETURN statement, the ENTRY statement, internal procedures and the CONTAINS statement,  
27 statement functions, generic procedure names, and non-Fortran procedures.
- 28 **Scoping and Association Rules.** Section 14 (Scope, Association, and Definition) explains the  
29 use of the term "scope" (especially important now because of the addition of internal procedures,  
30 modules, and other new features), and describes the scope properties of various entities, including  
31 names and operators. Also described are the general rules governing procedure argument associa-  
32 tion, pointer association, and storage association. Finally, Section 14 describes the events that  
33 cause variables to become defined (have predictable values) and events that cause variables to  
34 become undefined.
- 35 **Drafting and Presentation.** The text of this standard was developed using style rules defined  
36 by ANSI. In order to accommodate the differing requirements of the ISO Directives on Drafting  
37 and Presentation of International Standards, certain changes have been incorporated. Among  
38 those that have not been made is the form of the decimal sign, which in this standard is a period,  
39 as in 3.14159.



2 Subcommittee X3J3 on Fortran, with the guidance of the international Fortran Working Group  
3 ISO/IEC JTC1/SC22/WG5, developed this standard. The technical development has been carried  
4 out by subgroups, whose work is reviewed by the full committee. During the period of develop-  
5 ment of the draft Fortran standard, many persons assumed important roles of leadership:

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\*  
John K. Reid, Secretary  
Jeanne T. Martin, Secretary\* (1982-1987), Convener ISO/IEC JTC1/SC22/WG5  
Loren P. Meissner, Secretary (1978-1982)  
E. Andrew Johnson, International Representative\*  
Frances E. Holberton, International Representative (1978-1982)  
Neldon H. Marshall, Librarian\*  
Kurt W. Hirschert, Vocabulary Representative\*  
James H. Matheny, Vocabulary Representative\* (1986-1987)

#### SUBGROUP HEADS

Richard A. Hendrickson	(Brian T. Smith)
Kurt W. Hirschert	(Alan Wilson)
James H. Matheny	(Robert Allison)
Richard R. Ragan	(J. Lawrence Schonfelder)
E. Andrew Johnson	(Kevin W. Harris)
Lloyd W. Campbell	(Michael Metcalf)
Carl D. Burch	(Ivor R. Philips)



## FOREWORD

Those who contributed to the work by attending four or more meetings (X3J3) or three or more meetings (WG5) were:

Robert Allison  
Cornelis G. F. Ampt  
Stuart L. Anderson  
Charles Arnold  
Graham Barber  
John L. Barney  
Gloria M. Bauer\*  
Michael J. A. Berry  
Valerie G. Bowe  
Joanne Brixius  
Neil Brutman  
Albert Buckley  
Larry Bumgarner  
Carl D. Burch\*  
Winfried A. Burke\*  
Gary Campbell  
John H. Carman  
T. C. Chao  
Nancy Cheng  
P. Alan Clarke  
Joel Clinkenbeard  
Malcolm Cohen  
Joe Cointment  
Theodore R. Crowley  
Jeremy Du Croz  
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\*  
Leo G. J. ter Haar  
Kevin W. Harris  
Richard A. Hendrickson\*  
Dean A. Herington\*  
Maureen B. Smith Hoffert  
Tracy Ann Hoover  
Sheryl Horowitz  
Steve K. Hue  
Jagmohan L. Humar  
Gregory Johnson  
Peter N. Karculias  
Henry S. Katz  
Richard P. Kelble  
Leslie M. Klein  
Wilfried Kneis  
Paul St. Pierre

Werner Koblitz  
George T. Komorowski  
Joseph A. Korty  
Denise A. Lagasse  
Thomas M. Lahey  
Anil K. Lakhwara  
Sharon Lammers  
Dorothy E. Lang  
John E. Lauer\*  
Herrick S. Lauson  
Kay Leonard  
William Leonard  
Paul C. Libassi  
Donald L. Loe  
Warren E. Loper  
Bruce A. Martin\*  
Alex L. Marusak  
Christian J. Mas  
Evelyn S. B. Mast  
John Mayer  
Edward H. McCall  
Keith McConnell  
Brian L. Meek  
Michael Metcalf  
Fausto Milinazzo  
Geoff Millard  
Robert M. Miller  
J. S. Morgan  
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  
David Phillipmore  
Klaus Plasser  
Aurelio A. Pollicini  
Bruce W. Puerling\*  
Richard R. Ragan\*  
Lawrence Rolison  
Karl-Heinz Rotthausen  
Steven M. Rowan  
Paul St. Pierre  
Werner Schenk\*  
Gerhard J. Schmitt  
J. Lawrence Schonfelder  
Rick N. Schubert  
John C. Schwebel  
Mok-Kong Shen

Richard Shepardson  
Richard W. Signor\*  
Paul Sinclair  
Brian T. Smith\*  
Presley Smith  
Jan A. M. Snoek  
Hieronymus Sobiesiak  
Ken Sperka  
Bruce Stowell  
Sylvia Sund  
Mario Surdi  
Richard C. Swift  
Andrew D. Tait  
Brian L. Thompson  
Julian Tilbury  
Christian Ullrich  
Robert B. Upshaw\*  
David M. Vallance  
Nico Vossenstijn  
Hideo Wada  
Wolfgang Walter  
Graham Warren  
Richard W. Weaver  
George E. Weekly  
Bruce Weinman  
Everett H. Whitley  
Stanley J. Whitlock  
Gunter Wiesner  
Edward J. Wilkens  
Alan Wilson  
John D. Wilson  
Tammy Yan

\*Subgroup Head





# TABLE OF CONTENTS

FOREWORD .....	i
1. SCOPE OF STANDARD .....	1-1
1.1 Purpose.....	1-1
1.2 Processor .....	1-1
1.3 Inclusions and Exclusions.....	1-1
1.3.1 Inclusions .....	1-1
1.3.2 Exclusions .....	1-1
1.4 Conformance.....	1-1
1.4.1 FORTRAN 77 Compatibility .....	1-2
1.5 Notation Used in This Standard .....	1-3
1.5.1 Syntax Rules .....	1-3
1.5.2 Assumed Syntax Rules.....	1-4
1.5.3 Syntax Conventions and Characteristics.....	1-4
1.5.4 Text Conventions .....	1-4
1.6 Deleted and Obsolete Features.....	1-5
1.6.1 Nature of Deleted Features .....	1-5
1.6.2 Nature of Obsolete Features.....	1-5
1.7 Modules.....	1-5
1.8 Normative References .....	1-5
2. FORTRAN TERMS AND CONCEPTS .....	2-1
2.1 High Level Syntax.....	2-1
2.2 Program Unit Concepts.....	2-3
2.2.1 Executable Program.....	2-3
2.2.2 Main Program .....	2-3
2.2.3 Procedure .....	2-3
2.2.4 Module .....	2-4
2.3 Execution Concepts .....	2-4
2.3.1 Executable/Nonexecutable Statements.....	2-4
2.3.2 Statement Order.....	2-4
2.3.3 The END Statement .....	2-5
2.3.4 Execution Sequence .....	2-6
2.4 Data Concepts.....	2-6
2.4.1 Data Type.....	2-6
2.4.2 Data Value.....	2-6
2.4.3 Data Entity .....	2-6
2.4.4 Scalar.....	2-7
2.4.5 Array.....	2-7
2.4.6 Pointer .....	2-8
2.4.7 Storage.....	2-8
2.5 Fundamental Terms.....	2-8
2.5.1 Name and Designator .....	2-8
2.5.2 Keyword.....	2-8
2.5.3 Declaration.....	2-8
2.5.4 Definition .....	2-8
2.5.5 Reference.....	2-9
2.5.6 Association.....	2-9
2.5.7 Intrinsic.....	2-9
2.5.8 Operator .....	2-9
2.5.9 Sequence.....	2-9
3. CHARACTERS, LEXICAL TOKENS, AND SOURCE FORM .....	3-1

# TABLE OF CONTENTS

3.1	Processor Character Set.....	3-1
3.1.1	Letters.....	3-1
3.1.2	Digits.....	3-1
3.1.3	Underscore.....	3-1
3.1.4	Special Characters.....	3-1
3.1.5	Other Characters.....	3-2
3.2	Low-Level Syntax.....	3-2
3.2.1	Keywords.....	3-2
3.2.2	Names.....	3-2
3.2.3	Constants.....	3-2
3.2.4	Operators.....	3-3
3.2.5	Statement Labels.....	3-4
3.2.6	Delimiters.....	3-4
3.3	Source Form.....	3-4
3.3.1	Free Source Form.....	3-4
3.3.2	Fixed Source Form.....	3-6
3.4	Including Source Text.....	3-6
4.	INTRINSIC AND DERIVED DATA TYPES.....	4-1
4.1	The Concept of Data Type.....	4-1
4.1.1	Set of Values.....	4-1
4.1.2	Constants.....	4-1
4.1.3	Operations.....	4-1
4.2	Relationship of Types and Values to Objects.....	4-2
4.3	Intrinsic Data Types.....	4-2
4.3.1	Numeric Types.....	4-2
4.3.2	Nonnumeric Types.....	4-5
4.4	Derived Types.....	4-7
4.4.1	Derived-Type Definition.....	4-8
4.4.2	Determination of Derived Types.....	4-11
4.4.3	Derived-Type Values.....	4-12
4.4.4	Construction of Derived-Type Values.....	4-12
4.4.5	Derived-Type Operations and Assignment.....	4-12
4.5	Construction of Array Values.....	4-12
5.	DATA OBJECT DECLARATIONS AND SPECIFICATIONS.....	5-1
5.1	Type Declaration Statements.....	5-1
5.1.1	Type Specifiers.....	5-3
5.1.2	Attributes.....	5-5
5.2	Attribute Specification Statements.....	5-9
5.2.1	INTENT Statement.....	5-9
5.2.2	OPTIONAL Statement.....	5-10
5.2.3	Accessibility Statements.....	5-10
5.2.4	SAVE Statement.....	5-10
5.2.5	DIMENSION Statement.....	5-11
5.2.6	ALLOCATABLE Statement.....	5-11
5.2.7	POINTER Statement.....	5-11
5.2.8	TARGET Statement.....	5-12
5.2.9	DATA Statement.....	5-12
5.2.10	PARAMETER Statement.....	5-14
5.3	IMPLICIT Statement.....	5-14
5.4	NAMelist Statement.....	5-16
5.5	Storage Association of Data Objects.....	5-16

5.5.1	EQUIVALENCE Statement .....	5-17
5.5.2	COMMON Statement .....	5-18
6.	USE OF DATA OBJECTS.....	6-1
6.1	Scalars .....	6-2
6.1.1	Substrings.....	6-2
6.1.2	Structure Components .....	6-2
6.2	Arrays .....	6-3
6.2.1	Whole Arrays .....	6-3
6.2.2	Array Elements and Array Sections.....	6-3
6.3	Dynamic Association.....	6-6
6.3.1	ALLOCATE Statement.....	6-6
6.3.2	NULLIFY Statement.....	6-7
6.3.3	DEALLOCATE Statement .....	6-7
7.	EXPRESSIONS AND ASSIGNMENT .....	7-1
7.1	Expressions .....	7-1
7.1.1	Form of an Expression .....	7-1
7.1.2	Intrinsic Operations .....	7-4
7.1.3	Defined Operations .....	7-5
7.1.4	Data Type, Type Parameters, and Shape of an Expression .....	7-6
7.1.5	Conformability Rules for Intrinsic Operations.....	7-7
7.1.6	Scalar and Array Expressions .....	7-7
7.1.7	Evaluation of Operations.....	7-9
7.2	Interpretation of Intrinsic Operations .....	7-13
7.2.1	Numeric Intrinsic Operations .....	7-13
7.2.2	Character Intrinsic Operation .....	7-14
7.2.3	Relational Intrinsic Operations .....	7-14
7.2.4	Logical Intrinsic Operations .....	7-15
7.3	Interpretation of Defined Operations .....	7-16
7.3.1	Unary Defined Operation.....	7-16
7.3.2	Binary Defined Operation .....	7-16
7.4	Precedence of Operators .....	7-16
7.5	Assignment .....	7-18
7.5.1	Assignment Statement .....	7-18
7.5.2	Pointer Assignment .....	7-21
7.5.3	Masked Array Assignment—WHERE.....	7-21
8.	EXECUTION CONTROL.....	8-1
8.1	Executable Constructs Containing Blocks.....	8-1
8.1.1	Rules Governing Blocks.....	8-1
8.1.2	IF Construct .....	8-1
8.1.3	CASE Construct .....	8-3
8.1.4	DO Construct.....	8-5
8.2	Branching .....	8-11
8.2.1	Statement Labels .....	8-12
8.2.2	GO TO Statement.....	8-12
8.2.3	Computed GO TO Statement.....	8-12
8.2.4	ASSIGN and Assigned GO TO Statement .....	8-12
8.2.5	Arithmetic IF Statement.....	8-13
8.3	CONTINUE Statement.....	8-13
8.4	STOP Statement.....	8-13

**TABLE OF CONTENTS**

8.5	PAUSE Statement.....	8-13
9.	INPUT/OUTPUT STATEMENTS.....	9-1
9.1	Records.....	9-1
9.1.1	Formatted Record.....	9-1
9.1.2	Unformatted Record.....	9-1
9.1.3	Endfile Record.....	9-1
9.2	Files.....	9-2
9.2.1	External Files.....	9-2
9.2.2	Internal Files.....	9-4
9.3	File Connection.....	9-5
9.3.1	Unit Existence.....	9-5
9.3.2	Connection of a File to a Unit.....	9-5
9.3.3	Preconnection.....	9-6
9.3.4	The OPEN Statement.....	9-6
9.3.5	The CLOSE Statement.....	9-9
9.4	Data Transfer Statements.....	9-10
9.4.1	Control Information List.....	9-10
9.4.2	Data Transfer Input/Output List.....	9-13
9.4.3	Error, End-of-Record, and End-of-File Conditions.....	9-15
9.4.4	Execution of a Data Transfer Input/Output Statement.....	9-15
9.4.5	Printing of Formatted Records.....	9-18
9.4.6	Termination of Data Transfer Statements.....	9-18
9.5	File Positioning Statements.....	9-18
9.5.1	BACKSPACE Statement.....	9-19
9.5.2	ENDFILE Statement.....	9-19
9.5.3	REWIND Statement.....	9-19
9.6	File Inquiry.....	9-19
9.6.1	Inquiry Specifiers.....	9-20
9.6.2	Restrictions on Inquiry Specifiers.....	9-23
9.6.3	Inquire by Output List.....	9-23
9.7	Restrictions on Function References and List Items.....	9-23
9.8	Restriction on Input/Output Statements.....	9-23
10.	INPUT/OUTPUT EDITING.....	10-1
10.1	Explicit Format Specification Methods.....	10-1
10.1.1	FORMAT Statement.....	10-1
10.1.2	Character Format Specification.....	10-1
10.2	Form of a Format Item List.....	10-2
10.2.1	Edit Descriptors.....	10-2
10.2.2	Fields.....	10-3
10.3	Interaction Between Input/Output List and Format.....	10-3
10.4	Positioning by Format Control.....	10-4
10.5	Data Edit Descriptors.....	10-4
10.5.1	Numeric Editing.....	10-4
10.5.2	Logical Editing.....	10-9
10.5.3	Character Editing.....	10-9
10.5.4	Generalized Editing.....	10-9
10.6	Control Edit Descriptors.....	10-10
10.6.1	Position Editing.....	10-10
10.6.2	Slash Editing.....	10-11
10.6.3	Colon Editing.....	10-11
10.6.4	S, SP, and SS Editing.....	10-11

10.6.5	P Editing.....	10-11
10.6.6	BN and BZ Editing.....	10-12
10.7	Character String Edit Descriptors.....	10-12
10.7.1	Character Constant Edit Descriptor.....	10-12
10.7.2	H Editing.....	10-12
10.8	List-Directed Formatting .....	10-12
10.8.1	List-Directed Input .....	10-13
10.8.2	List-Directed Output .....	10-14
10.9	Namelist Formatting.....	10-15
10.9.1	Namelist Input .....	10-16
10.9.2	Namelist Output .....	10-18
11.	PROGRAM UNITS .....	11-1
11.1	Main Program.....	11-1
11.1.1	Main Program Specifications .....	11-1
11.1.2	Main Program Executable Part.....	11-1
11.1.3	Main Program Internal Procedures.....	11-2
11.2	External Subprograms.....	11-2
11.3	Modules.....	11-2
11.3.1	Module Reference .....	11-2
11.3.2	The USE Statement and Use Association .....	11-2
11.3.3	Examples of the Use of Modules .....	11-3
11.4	Block Data Program Units .....	11-6
12.	PROCEDURES.....	12-1
12.1	Procedure Classifications.....	12-1
12.1.1	Procedure Classification by Reference.....	12-1
12.1.2	Procedure Classification by Means of Definition.....	12-1
12.2	Characteristics of Procedures .....	12-2
12.2.1	Characteristics of Dummy Arguments.....	12-2
12.2.2	Characteristics of Function Results .....	12-3
12.3	Procedure Interface.....	12-3
12.3.1	Implicit and Explicit Interfaces.....	12-3
12.3.2	Specification of the Procedure Interface.....	12-4
12.4	Procedure Reference.....	12-8
12.4.1	Actual Argument List.....	12-8
12.4.2	Function Reference .....	12-10
12.4.3	Elemental Intrinsic Function Reference.....	12-10
12.4.4	Subroutine Reference .....	12-10
12.4.5	Elemental Intrinsic Subroutine Reference.....	12-11
12.5	Procedure Definition .....	12-11
12.5.1	Intrinsic Procedure Definition .....	12-11
12.5.2	Procedures Defined by Subprograms.....	12-11
12.5.3	Definition of Procedures by Means Other Than Fortran .....	12-17
12.5.4	Statement Function.....	12-17
13.	INTRINSIC PROCEDURES.....	13-1
13.1	Intrinsic Functions .....	13-1
13.2	Elemental Intrinsic Procedures .....	13-1
13.2.1	Elemental Intrinsic Function Arguments and Results .....	13-1
13.2.2	Elemental Intrinsic Subroutine Arguments .....	13-1
13.3	Positional Arguments or Argument Keywords .....	13-1

**TABLE OF CONTENTS**

13.4	Argument Presence Inquiry Function.....	13-1
13.5	Numeric, Mathematical, Character, Kind, Logical, and Bit Procedures.....	13-1
13.5.1	Numeric Functions .....	13-1
13.5.2	Mathematical Functions.....	13-2
13.5.3	Character Functions.....	13-2
13.5.4	Character Inquiry Function.....	13-2
13.5.5	Kind Functions .....	13-2
13.5.6	Logical Function.....	13-2
13.5.7	Bit Manipulation and Inquiry Procedures .....	13-2
13.6	Transfer Function.....	13-2
13.7	Numeric Manipulation and Inquiry Functions .....	13-3
13.7.1	Models for Integer and Real Data .....	13-3
13.7.2	Numeric Inquiry Functions.....	13-3
13.7.3	Floating Point Manipulation Functions.....	13-3
13.8	Array Intrinsic Functions.....	13-3
13.8.1	The Shape of Array Arguments.....	13-3
13.8.2	Mask Arguments .....	13-4
13.8.3	Vector and Matrix Multiplication Functions .....	13-4
13.8.4	Array Reduction Functions .....	13-4
13.8.5	Array Inquiry Functions .....	13-4
13.8.6	Array Construction Functions .....	13-4
13.8.7	Array Reshape Function .....	13-4
13.8.8	Array Manipulation Functions .....	13-4
13.8.9	Array Location Functions .....	13-5
13.8.10	Pointer Association Status Inquiry Functions .....	13-5
13.9	Intrinsic Subroutines.....	13-5
13.9.1	Date and Time Subroutines.....	13-5
13.9.2	Pseudorandom Numbers.....	13-5
13.9.3	Bit Copy Subroutine .....	13-5
13.10	Generic Intrinsic Functions.....	13-5
13.10.1	Argument Presence Inquiry Function .....	13-5
13.10.2	Numeric Functions .....	13-5
13.10.3	Mathematical Functions.....	13-6
13.10.4	Character Functions.....	13-6
13.10.5	Character Inquiry Function.....	13-7
13.10.6	Kind Functions .....	13-7
13.10.7	Logical Function.....	13-7
13.10.8	Numeric Inquiry Functions.....	13-7
13.10.9	Bit Inquiry Function .....	13-7
13.10.10	Bit Manipulation Functions.....	13-7
13.10.11	Transfer Function.....	13-7
13.10.12	Floating-point Manipulation Functions .....	13-7
13.10.13	Vector and Matrix Multiply Functions.....	13-8
13.10.14	Array Reduction Functions .....	13-8
13.10.15	Array Inquiry Functions .....	13-8
13.10.16	Array Construction Functions .....	13-8
13.10.17	Array Reshape Function .....	13-8
13.10.18	Array Manipulation Functions .....	13-9
13.10.19	Array Location Functions .....	13-9
13.10.20	Pointer Association Status Inquiry Function.....	13-9
13.11	Intrinsic Subroutines.....	13-9
13.12	Specific Names for Intrinsic Functions .....	13-9
13.13	Specifications of the Intrinsic Procedures.....	13-11
14.	SCOPE, ASSOCIATION, AND DEFINITION.....	14-1

14.1	Scope of Names .....	14-1
14.1.1	Global Entities .....	14-1
14.1.2	Local Entities .....	14-1
14.1.3	Statement Entities .....	14-4
14.2	Scope of Labels .....	14-5
14.3	Scope of External Input/Output Units.....	14-5
14.4	Scope of Operators.....	14-5
14.5	Scope of the Assignment Symbol .....	14-5
14.6	Association.....	14-5
14.6.1	Name Association.....	14-5
14.6.2	Pointer Association.....	14-6
14.6.3	Storage Association .....	14-6
14.7	Definition and Undefined of Variables.....	14-8
14.7.1	Definition of Objects and Subobjects .....	14-8
14.7.2	Variables That Are Always Defined .....	14-8
14.7.3	Variables That Are Initially Defined.....	14-9
14.7.4	Variables That Are Initially Undefined .....	14-9
14.7.5	Events That Cause Variables to Become Defined .....	14-9
14.7.6	Events That Cause Variables to Become Undefined.....	14-10
14.8	Allocation Status .....	14-11
A.	GLOSSARY OF TECHNICAL TERMS .....	A-1
B.	DECREMENTAL FEATURES .....	B-1
B.1	Deleted Features.....	B-1
B.2	Obsolescent Features .....	B-1
B.2.1	Alternate Return .....	B-1
B.2.2	PAUSE Statement .....	B-1
B.2.3	ASSIGN and Assigned GO TO Statements.....	B-1
B.2.4	Assigned FORMAT Specifiers .....	B-2
B.2.5	H editing .....	B-2
C.	SECTION NOTES .....	C-1
C.1	Section 1 Notes .....	C-1
C.1.1	Conformance (1.4).....	C-1
C.2	Section 2 Notes .....	C-1
C.2.1	Keywords .....	C-1
C.3	Section 3 Notes .....	C-1
C.3.1	Representable Characters (3.1.5).....	C-1
C.3.2	Comment Lines (3.3.1.1, 3.3.2.1).....	C-1
C.3.3	Statement Labels (3.2.5).....	C-1
C.3.4	Source Form (3.3) .....	C-1
C.4	Section 4 Notes .....	C-2
C.4.1	Zero (4.3.1) .....	C-2
C.4.2	Characters (4.2).....	C-2
C.4.3	Intrinsic and Derived Data Types (4.3, 4.4).....	C-2
C.4.4	Selection of the Approximation Methods .....	C-3
C.4.5	Storage of Derived Types (4.4.1).....	C-4
C.4.6	Pointers.....	C-4
C.5	Section 5 Notes .....	C-5
C.5.1	Type Declaration Statements (5.1).....	C-5
C.5.2	The POINTER Attribute (5.1.2.7) .....	C-5
C.5.3	The TARGET Attribute (5.1.2.8) .....	C-5

**TABLE OF CONTENTS**

	C.5.4	PARAMETER Statements and IMPLICIT NONE (5.2.10, 5.3) .....	C-6
	C.5.5	EQUIVALENCE Statement Extensions (5.5.1) .....	C-6
	C.5.6	COMMON Statement Extensions (5.5.2).....	C-6
C.6		Section 6 Notes .....	C-7
	C.6.1	Substrings (6.1.1).....	C-7
	C.6.2	Array Element References (6.2.2) .....	C-7
	C.6.3	Structure Components (6.1.2).....	C-7
	C.6.4	Pointer Allocation and Association.....	C-8
	C.6.5	Partial Summary of Array Name Appearances .....	C-9
C.7		Section 7 Notes .....	C-9
	C.7.1	Character Assignment.....	C-9
	C.7.2	Evaluation of Function References.....	C-9
	C.7.3	Pointers in Expressions .....	C-9
	C.7.4	Pointers on the Left Side of an Assignment .....	C-10
C.8		Section 8 Notes .....	C-10
	C.8.1	Loop Control .....	C-10
	C.8.2	The CASE Construct.....	C-11
	C.8.3	Examples of Invalid DO Constructs.....	C-11
C.9		Section 9 Notes .....	C-11
	C.9.1	Input/Output Records (9.1) .....	C-11
	C.9.2	Files (9.2).....	C-11
	C.9.3	OPEN Statement (9.3.4).....	C-13
	C.9.4	Connection Properties (9.3.2) .....	C-14
	C.9.5	CLOSE Statement (9.3.5).....	C-15
	C.9.6	INQUIRE Statement (9.6) .....	C-16
	C.9.7	Keyword Specifiers.....	C-16
	C.9.8	Format Specifications (9.4.1.1).....	C-16
	C.9.9	Unformatted Input/Output (9.4.4.1) .....	C-17
	C.9.10	Input/Output Restrictions .....	C-17
	C.9.11	Pointers in an Input/Output List .....	C-17
	C.9.12	Derived Type Objects in an Input/Output List (9.4.2).....	C-17
C.10		Section 10 Notes .....	C-17
	C.10.1	Character Constant Format Specification (10.1.2, 10.7.1).....	C-17
	C.10.2	T Edit Descriptor (10.6.1.1) .....	C-18
	C.10.3	Length of Formatted Records .....	C-18
	C.10.4	Number of Records (10.3, 10.4, 10.6.2).....	C-18
	C.10.5	List-Directed Input/Output (10.8) .....	C-18
	C.10.6	List-Directed Input (10.8.1).....	C-18
	C.10.7	Namelist List Items for Character Input (10.9.1.3) .....	C-19
	C.10.8	Namelist Output Records (10.9.2.2) .....	C-19
C.11		Section 11 Notes .....	C-19
	C.11.1	Main Program and Block Data Program Unit (11.1, 11.4) .....	C-19
	C.11.2	Examples of Host Association (11.2.2).....	C-19
	C.11.3	Dependent Compilation (11.3).....	C-20
	C.11.4	Pointers in Modules.....	C-22
	C.11.5	Example of a Module (11.3).....	C-22
C.12		Section 12 Notes .....	C-25
	C.12.1	External Procedures (12.3.2.2).....	C-25
	C.12.2	Procedures Defined by Means Other Than Fortran (12.5.3).....	C-25
	C.12.3	Procedure Interfaces (12.3) .....	C-26
	C.12.4	Argument Association and Evaluation (12.4.1).....	C-26
	C.12.5	Argument Intent Specification (12.4.1.1) .....	C-27
	C.12.6	Dummy Argument Restrictions (12.5.2.9).....	C-27
	C.12.7	Pointers and Targets as Arguments .....	C-27



	C.12.8	The ASSOCIATED Function (13.13.13) .....	C-28
	C.12.9	Internal Procedure Restrictions .....	C-28
	C.12.10	The Result Variable (12.5.2.2) .....	C-28
C.13	Section 13	Notes .....	C-29
	C.13.1	Summary of Features .....	C-29
	C.13.2	Examples .....	C-30
	C.13.3	FORMula TRANslation and Array Processing .....	C-34
	C.13.4	Sum of Squared Residuals .....	C-35
	C.13.5	Vector Norms: Infinity-Norm and One-Norm .....	C-35
	C.13.6	Matrix Norms: Infinity-Norm and One-Norm .....	C-35
	C.13.7	Logical Queries.....	C-35
	C.13.8	Parallel Computations.....	C-36
	C.13.9	Example of Element-by-Element Computation .....	C-36
	C.13.10	Bit Manipulation and Inquiry Procedures .....	C-37
C.14	Section 14	Notes .....	C-37
	C.14.1	Unambiguous Generic Procedure References (14.1.2.3) .....	C-37
	C.14.2	Storage Association of Zero-Sized Objects .....	C-38
D.	SYNTAX RULES .....		D-1
D.1	Syntax Rules and Constraints .....		D-1
	D.1.1	Scope of Standard .....	D-1
	D.1.2	Fortran Terms and Concepts.....	D-1
	D.1.3	Characters, Lexical Tokens, and Source Form.....	D-3
	D.1.4	Intrinsic and Derived Data Types .....	D-5
	D.1.5	Data Object Declarations and Specifications .....	D-7
	D.1.6	Use of Data Objects.....	D-12
	D.1.7	Expressions and Assignment .....	D-14
	D.1.8	Execution Control .....	D-17
	D.1.9	Input/Output Statements.....	D-20
	D.1.10	Input/Output Editing .....	D-23
	D.1.11	Program Units .....	D-24
	D.1.12	Procedures .....	D-25
	D.1.13	Intrinsic Procedures.....	D-28
	D.1.14	Scope, Association, and Definition.....	D-28
D.2	Cross References .....		D-28
	D.2.1	Nonterminal Symbols That Are Defined .....	D-28
	D.2.2	Nonterminal Symbols That Are Not Defined.....	D-34
	D.2.3	Terminal Symbols .....	D-34
E.	PERMUTED INDEX FOR HEADINGS .....		E-1
F.	INDEX.....		F-1

FIGURES

Figure 2.1	Requirements on Statement Ordering .....	2-5
------------	--	-----

TABLES

Table 2.1	Statements Allowed in Scoping Units .....	2-5
Table 6.1	Subscript Order Value .....	6-4
Table 7.1	Type of Operands and Result for the Intrinsic Operation $[x_1] \text{ op } x_2$ .....	7-5
Table 7.2	Interpretation of the Numeric Intrinsic Operators .....	7-13
Table 7.3	Interpretation of the Character Intrinsic Operator // .....	7-14

## TABLE OF CONTENTS

Table 7.4 Interpretation of the Relational Intrinsic Operators.....	7-14
Table 7.5 Interpretation of the Logical Intrinsic Operators.....	7-15
Table 7.6 The Values of Operations Involving Logical Intrinsic Operators .....	7-15
Table 7.7 Categories of Operations and Relative Precedences .....	7-16
Table 7.8 Type Conformance for the Intrinsic Assignment Statement <i>variable = expr</i> .....	7-18
Table 7.9 Numeric Conversion and Assignment Statement <i>variable = expr</i> .....	7-19
Table C.1 Some of the Possible Appearances of Array Names .....	C-9
Table C.2 Values Assigned to INQUIRE Specifier Variables.....	C-16

# 1. SCOPE OF STANDARD

1 **1.1 Purpose.** This standard specifies the form and establishes the interpretation of programs  
2 expressed in the Fortran language. The purpose of this standard is to promote portability, reliabil-  
3 ity, maintainability, and efficient execution of Fortran programs for use on a variety of computing  
4 systems. The foreword and appendices are not part of this standard.

5 **1.2 Processor.** The combination of a computing system and the mechanism by which programs  
6 are transformed for use on that computing system is called a processor in this standard.

7 **1.3 Inclusions and Exclusions.** This standard specifies the bounds of the Fortran language by  
8 identifying both those items included and those items excluded.

9 **1.3.1 Inclusions.** This standard specifies:

- 10 (1) The forms that a program written in the Fortran language may take
- 11 (2) The rules for interpreting the meaning of a program and its data
- 12 (3) The form of the input data to be processed by such a program
- 13 (4) The form of the output data resulting from the use of such a program

14 **1.3.2 Exclusions.** This standard does not specify:

- 15 (1) The mechanism by which programs are transformed for use on computing systems
- 16 (2) The operations required for setup and control of the use of programs on computing sys-  
17 tems
- 18 (3) The method of transcription of programs or their input or output data to or from a stor-  
19 age medium
- 20 (4) The program and processor behavior when the rules of this standard fail to establish an  
21 interpretation except for the processor detection and reporting requirements in items (2)  
22 to (8) of 1.4
- 23 (5) The size or complexity of a program and its data that will exceed the capacity of any  
24 specific computing system or the capability of a particular processor
- 25 (6) The physical properties of the representation of quantities and the method of rounding,  
26 approximating, or computing numeric values on a particular processor
- 27 (7) The physical properties of input/output records, files, and units
- 28 (8) The physical properties and implementation of storage

29 **1.4 Conformance.** The requirements, prohibitions, and options specified in this standard refer  
30 primarily to permissible forms and relationships for a standard-conforming program rather than  
31 for a processor. The optional output forms produced by a processor, which are not under the con-  
32 trol of a program, are an example of an exception. The requirements, prohibitions, and options for  
33 a standard-conforming processor usually must be inferred from those given for programs.

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

38 A processor conforms to this standard if:

- 39 (1) It executes any standard-conforming program in a manner that fulfills the interpreta-  
40 tions herein, subject to any limits that the processor may impose on the size and com-  
41 plexity of the program.

## Scope of Standard

- 1 (2) It contains the capability to detect and report the use within a submitted program unit  
2 of a form designated herein as deleted or obsolescent, insofar as such use can be  
3 detected by reference to the numbered syntax rules and their associated constraints.
- 4 (3) It contains the capability to detect and report the use within a submitted program unit  
5 of an additional form or relationship that is not permitted by the numbered syntax rules  
6 or their associated constraints.
- 7 (4) It contains the capability to detect and report the use within a submitted program of  
8 kind type parameter values (4.3) not supported by the processor.
- 9 (5) It contains the capability to detect and report the use within a submitted program of  
10 source form or characters not permitted by Section 3.
- 11 (6) It contains the capability to detect and report the use within a submitted program of  
12 name usage not consistent with the scope rules for names, labels, operators, and assign-  
13 ment symbols in Section 14.
- 14 (7) It contains the capability to detect and report the use within a submitted program of  
15 intrinsic procedures whose names are not defined in Section 13.
- 16 (8) It contains the capability to detect and report the reason for rejecting a program.

17 However, in a *format-specification* that is not part of a *format-stmt* (10.1.1), a processor is not  
18 required to detect or report the use of deleted or obsolescent features, or the use of additional  
19 forms or relationships.

20 A standard-conforming processor may allow additional forms and relationships provided that  
21 such additions do not conflict with the standard forms and relationships. However, a standard-  
22 conforming processor may allow additional intrinsic procedures even though this could cause a  
23 conflict with the name of a procedure in a standard-conforming program. If such a conflict occurs  
24 and involves the name of an external procedure, the processor is permitted to use the intrinsic pro-  
25 cedure unless the name is given, the EXTERNAL attribute in the same scoping unit (14). A  
26 standard-conforming program must not use nonstandard intrinsic procedures that have been  
27 added by the processor.

28 Note that a standard-conforming program must not use any forms or relationships that are prohib-  
29 ited by this standard, but a standard-conforming processor may allow such forms and relation-  
30 ships if they do not change the proper interpretation of a standard-conforming program. For  
31 example, a standard-conforming processor may allow a nonstandard data type.

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

37 In some cases, this standard allows the provision of facilities that are not completely specified in  
38 the standard. These facilities are identified as **processor dependent**. The facility must be pro-  
39 vided, with methods or semantics determined by the processor.

40 **1.4.1 FORTRAN 77 Compatibility.** Except as noted in this section, this standard is an upward com-  
41 patible extension to the preceding Fortran standard, ISO 1539-1980, informally referred to as FOR-  
42 TRAN 77 and a standard-conforming processor for this standard is a standard-conforming processor  
43 for FORTRAN 77. Any standard-conforming FORTRAN 77 program remains standard conforming  
44 under this standard; however, see item (4) below regarding intrinsic procedures. This standard  
45 restricts the behavior for some features that were processor dependent in FORTRAN 77. Therefore, a  
46 standard-conforming FORTRAN 77 program that uses one of these processor-dependent features  
47 may have a different interpretation under this standard, yet remain a standard-conforming pro-  
48 gram. The following FORTRAN 77 features have different interpretations in this standard:

- 49 (1) FORTRAN 77 permitted a processor to supply more precision derived from a real constant  
50 than can be contained in a real datum when the constant is used to initialize a DOUBLE

- 1           PRECISION data object in a DATA statement. This standard does not permit a proces-  
2           sor this option.
- 3           (2) If a named variable that is not in a common block is initialized in a DATA statement  
4           and does not have the SAVE attribute specified, FORTRAN 77 left its SAVE attribute proces-  
5           sor dependent. This standard specifies that this named variable has the SAVE attri-  
6           bute (5.2.9).
- 7           (3) FORTRAN 77 required that the number of values required by the input list must be less  
8           than or equal to the number of values in the record during formatted input. This stand-  
9           ard specifies that the input record is logically padded with blanks if there are not  
10          enough values in the record, unless the PAD= 'NO' option is specified in an appropriate  
11          OPEN statement (9.4.4.4.2)
- 12          (4) This standard has more intrinsic functions than did FORTRAN 77 and adds a few intrinsic  
13          subroutines. Therefore, a standard-conforming FORTRAN 77 program may have a differ-  
14          ent interpretation under this standard if it invokes a procedure having the same name  
15          as one of the new standard intrinsic procedures, unless that procedure is specified in an  
16          EXTERNAL statement as recommended for nonintrinsic functions in the appendix to  
17          the FORTRAN 77 standard.

18   **1.5 Notation Used in This Standard.** In this standard, "must" is to be interpreted as a  
19   requirement; conversely, "must not" is to be interpreted as a prohibition.

20   **1.5.1 Syntax Rules.** Syntax rules are used to help describe the form that Fortran lexical tokens,  
21   statements, and constructs may take. These syntax rules are expressed in a variation of Backus-  
22   Naur form (BNF) in which:

- 23          (1) Characters from the Fortran character set (3.1) are to be written as shown, except where  
24          otherwise noted.
- 25          (2) Lower-case italicized letters and words (often hyphenated and abbreviated) represent  
26          general syntactic classes for which specific syntactic entities must be substituted in  
27          actual statements.

28          Some common abbreviations used in syntactic terms are:

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

- 34          (3) The syntactic metasympols used are:

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

- 41          (4) Each syntax rule is given a unique identifying number of the form *R<sub>snn</sub>*, where *s* is a  
42          one- or two-digit section number and *nn* is a two-digit sequence number within that  
43          section. The syntax rules are distributed as appropriate throughout the text, and are  
44          referenced by number as needed. Some rules in Sections 2 and 3 are more fully  
45          described in later sections; in such cases, the section number *s* is the number of the later  
46          section where the rule is repeated. The rules also are collected in Appendix D.

## Scope of Standard

- 1 (5) The syntax rules are not a complete and accurate syntax description of Fortran, and cannot  
2 be used to generate automatically a Fortran parser; where a syntax rule is incom-  
3 plete, it is accompanied by the corresponding constraints.
- 4 (6) Obsolescent features (1.6) are shown in a distinguishing type size. This is an example of the  
5 size used for obsolescent features.

6 An example of the use of syntax rules is:

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

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

9 *digit*  
10 *digit digit*  
11 *digit digit digit digit*  
12 *digit digit digit digit digit digit digit digit*

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

14 4  
15 67  
16 1999  
17 10243852

18 **1.5.2 Assumed Syntax Rules.** In order to minimize the number of additional syntax rules and  
19 convey appropriate constraint information, the following rules are assumed. The letters "*xyz*"  
20 stand for any legal syntactic class phrase:

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

22 *xyz-name* is *name*

23 *scalar-xyz* is *xyz*

24 Constraint: *scalar-xyz* must be scalar.

### 25 1.5.3 Syntax Conventions and Characteristics.

- 26 (1) Any syntactic class name ending in "*-stmt*" follows the source form statement rules: it  
27 must be delimited by end-of-line or semicolon, and may be labeled unless it forms part  
28 of another statement (such as an IF or WHERE statement). Conversely, everything con-  
29 sidered to be a source form statement is given a "*-stmt*" ending in the syntax rules.
- 30 (2) The rules on statement ordering are described rigorously in the definition of *program-*  
31 *unit* (R202-R216). Expression hierarchy is described rigorously in the definition of *expr*  
32 (R723).
- 33 (3) The suffix "*-spec*" is used consistently for specifiers, such as keyword actual arguments  
34 and input/output statement specifiers. It also is used for type declaration attribute  
35 specifications (for example, "*array-spec*" in R512), and in a few other cases.
- 36 (4) When reference is made to a type parameter, including the surrounding parentheses,  
37 the term "*selector*" is used. See, for example, "*length-selector*" (R507) and "*kind-selector*"  
38 (R505).
- 39 (5) The term "*subscript*" (for example, R617, R618, and R619) is used consistently in array  
40 definitions.

41 **1.5.4 Text Conventions.** In the descriptive text, the normal English word equivalent of a BNF  
42 syntactic term is usually used. Specific statements and attributes are identified in the text by the  
43 upper-case keyword, e.g., "END statement". Boldface words are used in the text where they are  
44 first defined with a specialized meaning.

1 **1.6 Deleted and Obsolescent Features.** This standard protects the users' investment in  
2 existing software by including all of the language elements of FORTRAN 77 that are not processor  
3 dependent. This document identifies two categories of outmoded features. There are none in the  
4 first category, **deleted features**, which consists of features considered to have been redundant in  
5 FORTRAN 77 and largely unused. Those in the second category, **obsolescent features**, are consid-  
6 ered to have been redundant in FORTRAN 77, but are still used frequently.

7 **1.6.1 Nature of Deleted Features.**

- 8 (1) Better methods existed in FORTRAN 77.  
9 (2) These features are not included in this revision of Fortran.

10 **1.6.2 Nature of Obsolescent Features.**

- 11 (1) Better methods existed in FORTRAN 77.  
12 (2) It is recommended that programmers use these better methods in new programs and  
13 convert existing code to these methods.  
14 (3) These features are identified in the text of this document by a distinguishing type font  
15 (1.5.1).  
16 (4) If the use of these features has become insignificant in Fortran programs, it is recom-  
17 mended that future Fortran standards committees consider deleting them from the next  
18 revision.  
19 (5) It is recommended that the next Fortran standards committee consider for deletion only  
20 those language features that appear in the list of obsolescent features.  
21 (6) It is recommended that processors supporting the Fortran language continue to support  
22 these features as long as they continue to be used widely in Fortran programs.

23 **1.7 Modules.** This standard provides facilities that encourage the design and use of modular  
24 and reusable software. Data and procedure definitions may be organized into nonexecutable pro-  
25 gram units, called modules, and made available to any other program unit. In addition to global  
26 data and procedure library facilities, modules provide a mechanism for defining data abstractions  
27 and certain language extensions. Modules are described in 11.3.

28 A module may be standardized as a separate collateral standard. A **standard module** must not  
29 use any obsolescent feature, nor any nonstandard form or relationship.

30 **1.8 Normative References.** The following standards contain provisions which, through ref-  
31 erence in this text, constitute provisions of the International Standard. At the time of publication,  
32 the editions indicated were valid. All standards are subject to revision, and parties to agreements  
33 based upon this International Standard are encouraged to investigate the possibility of applying  
34 the most recent editions of the standards indicated below. Members of IEC and ISO maintain reg-  
35 isters of currently valid International Standards.

- 36 ISO 8601:1988 *Data Elements and Interchange Formats*  
37 ANSI X3.4-1986 (ASCII)  
38 CCIR Recommendation 460-2





## 2. FORTRAN TERMS AND CONCEPTS

1 **2.1 High Level Syntax.** This section introduces the terms associated with program units and  
2 other Fortran concepts above the construct, statement, and expression levels and illustrates their  
3 relationships. The syntax rule notation is described in 1.5.1. Note that some of the syntax rules in  
4 this section are subject to constraints that are given only at the appropriate places in later sections.

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

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

8 R202 *program-unit* is *main-program*  
9 or *external-subprogram*  
10 or *module*  
11 or *block-data*

12 R1101 *main-program* is [ *program-stmt* ]  
13 [ *specification-part* ]  
14 [ *execution-part* ]  
15 [ *internal-subprogram-part* ]  
16 *end-program-stmt*

17 R203 *external-subprogram* is *function-subprogram*  
18 or *subroutine-subprogram*

19 R1215 *function-subprogram* is *function-stmt*  
20 [ *specification-part* ]  
21 [ *execution-part* ]  
22 [ *internal-subprogram-part* ]  
23 *end-function-stmt*

24 R1219 *subroutine-subprogram* is *subroutine-stmt*  
25 [ *specification-part* ]  
26 [ *execution-part* ]  
27 [ *internal-subprogram-part* ]  
28 *end-subroutine-stmt*

29 R1104 *module* is *module-stmt*  
30 [ *specification-part* ]  
31 [ *module-subprogram-part* ]  
32 *end-module-stmt*

33 R1110 *block-data* is *block-data-stmt*  
34 [ *specification-part* ]  
35 *end-block-data-stmt*

36 R204 *specification-part* is [ *use-stmt* ] ...  
37 [ *implicit-part* ]  
38 [ *declaration-construct* ] ...

39 R205 *implicit-part* is [ *implicit-part-stmt* ] ...  
40 *implicit-stmt*

41 R206 *implicit-part-stmt* is *implicit-stmt*  
42 or *parameter-stmt*  
43 or *format-stmt*  
44 or *entry-stmt*

45 R207 *declaration-construct* is *derived-type-def*  
46 or *interface-block*  
47 or *type-declaration-stmt*  
48 or *specification-stmt*  
49 or *parameter-stmt*

## Fortran Terms and Concepts

1			or <i>format-stmt</i>
2			or <i>entry-stmt</i>
3			or <i>stmt-function-stmt</i>
4	R208	<i>execution-part</i>	is <i>executable-construct</i>
5			[ <i>execution-part-construct</i> ] ...
6	R209	<i>execution-part-construct</i>	is <i>executable-construct</i>
7			or <i>format-stmt</i>
8			or <i>data-stmt</i>
9			or <i>entry-stmt</i>
10	R210	<i>internal-subprogram-part</i>	is <i>contains-stmt</i>
11			<i>internal-subprogram</i>
12			[ <i>internal-subprogram</i> ] ...
13	R211	<i>internal-subprogram</i>	is <i>function-subprogram</i>
14			or <i>subroutine-subprogram</i>
15	R212	<i>module-subprogram-part</i>	is <i>contains-stmt</i>
16			<i>module-subprogram</i>
17			[ <i>module-subprogram</i> ] ...
18	R213	<i>module-subprogram</i>	is <i>function-subprogram</i>
19			or <i>subroutine-subprogram</i>
20	R214	<i>specification-stmt</i>	is <i>access-stmt</i>
21			or <i>allocatable-stmt</i>
22			or <i>common-stmt</i>
23			or <i>data-stmt</i>
24			or <i>dimension-stmt</i>
25			or <i>equivalence-stmt</i>
26			or <i>external-stmt</i>
27			or <i>intent-stmt</i>
28			or <i>intrinsic-stmt</i>
29			or <i>namelist-stmt</i>
30			or <i>optional-stmt</i>
31			or <i>pointer-stmt</i>
32			or <i>save-stmt</i>
33			or <i>target-stmt</i>
34	R215	<i>executable-construct</i>	is <i>action-stmt</i>
35			or <i>case-construct</i>
36			or <i>do-construct</i>
37			or <i>if-construct</i>
38			or <i>where-construct</i>
39	R216	<i>action-stmt</i>	is <i>allocate-stmt</i>
40			or <i>assignment-stmt</i>
41			or <i>backspace-stmt</i>
42			or <i>call-stmt</i>
43			or <i>close-stmt</i>
44			or <i>computed-goto-stmt</i>
45			or <i>continue-stmt</i>
46			or <i>cycle-stmt</i>
47			or <i>deallocate-stmt</i>
48			or <i>endfile-stmt</i>
49			or <i>end-function-stmt</i>
50			or <i>end-program-stmt</i>
51			or <i>end-subroutine-stmt</i>
52			or <i>exit-stmt</i>
53			or <i>goto-stmt</i>

1 or *if-stmt*  
 2 or *inquire-stmt*  
 3 or *nullify-stmt*  
 4 or *open-stmt*  
 5 or *pointer-assignment-stmt*  
 6 or *print-stmt*  
 7 or *read-stmt*  
 8 or *return-stmt*  
 9 or *rewind-stmt*  
 10 or *stop-stmt*  
 11 or *where-stmt*  
 12 or *write-stmt*  
 13 or *arithmetic-if-stmt*  
 14 or *assign-stmt*  
 15 or *assigned-goto-stmt*  
 16 or *pause-stmt*

17 **Constraint:** An *execution-part-construct* must not contain an *end-function-stmt*, *end-program-stmt*, or  
 18 *end-subroutine-stmt*.

19 **2.2 Program Unit Concepts.** Program units are the fundamental components of a Fortran  
 20 program. A program unit may be a main program, an external subprogram, a module, or a block  
 21 data program unit. A subprogram may be a function subprogram or a subroutine subprogram. A  
 22 module contains definitions that are to be made accessible to other program units. A block data  
 23 program unit is used to specify initial values for data objects in named common blocks. Each type  
 24 of program unit is described in Section 11 or 12. An **external subprogram** is a subprogram that is  
 25 not contained within a main program, a module, or another subprogram. An **internal subpro-**  
 26 **gram** is a subprogram that is contained within a main program or another subprogram. A **module**  
 27 **subprogram** is a subprogram that is contained in a module but is not an internal subprogram.

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

- 29 (1) A derived-type definition (4.4.1),  
 30 (2) A procedure interface body, excluding any derived-type definitions and procedure  
 31 interface bodies contained within it (12.3.2.1), or  
 32 (3) A program unit or subprogram, excluding derived-type definitions, procedure interface  
 33 bodies, and subprograms contained within it.

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

35 **2.2.1 Executable Program.** An executable program consists of exactly one main program unit  
 36 and any number (including zero) of other kinds of program units. The set of program units may  
 37 include any combination of the different kinds of program units in any order.

38 **2.2.2 Main Program.** The main program is described in 11.1.

39 **2.2.3 Procedure.** A procedure encapsulates an arbitrary sequence of computations that may be  
 40 invoked directly during program execution. ~~A principal difference between the two kinds of pro-~~  
 41 ~~cedures is the way in which each is invoked.~~ A **function** is a procedure that is invoked in an  
 42 expression; its invocation causes a value to be computed which is then used in evaluating the  
 43 expression. The variable that returns the value of a function is called the **result variable**. A **sub-**  
 44 **routine** is a procedure that is invoked in a CALL statement or by a defined assignment statement  
 45 (12.4.4, 7.5.1.3). A subroutine may be used to change the program state by changing the values of  
 46 any of the data objects accessible to the subroutine; a function may do this in addition to comput-  
 47 ing the function value.

48 Procedures are described further in Section 12.

Procedures are either functions or subroutines.

need  
2-4-9-10  
have also

- 1 **2.2.3.1 External Procedure.** An external procedure is a procedure that is defined by an external  
2 subprogram or by means other than Fortran. An external procedure may be invoked by the main  
3 program or by any procedure of an executable program.
- 4 **2.2.3.2 Module Procedure.** A module procedure is a procedure that is defined by a module  
5 subprogram (R213). A module procedure may be invoked by another module subprogram in the  
6 module or by any scoping unit using the module. The module containing the subprogram is called  
7 the host of the module procedure.
- 8 **2.2.3.3 Internal Procedure.** An internal procedure is a procedure that is defined by an internal  
9 subprogram (R211). The containing main program or subprogram is called the host of the internal  
10 procedure. An internal procedure is local to its host in the sense that the internal procedure is  
11 accessible within the scoping units of the host and all its other internal procedures but is not acces-  
12 sible elsewhere.
- 13 **2.2.3.4 Procedure Interface Block.** The purpose of a procedure interface block is to describe  
14 the interfaces (12.3) to a set of procedures and to permit them to be invoked through either a single  
15 generic name, a defined operator, or a defined assignment. It determines the forms of reference  
16 through which the procedure may be invoked.
- 17 **2.2.4 Module.** A module contains (or accesses from other modules) definitions that are to be  
18 made accessible to other program units. These definitions include data object declarations, type  
19 definitions, procedure definitions, and procedure interface blocks. The purpose of a module is to  
20 make the definitions it contains accessible to all other program units in an executable program that  
21 request such accessibility. A scoping unit in another program unit may request access to the defi-  
22 nitions contained in a module. Modules are further described in Section 11.
- 23 **2.3 Execution Concepts.** A program unit is a sequence of statements. Each statement is clas-  
24 sified as either an executable statement or a nonexecutable statement. There are restrictions on  
25 the order in which statements may appear in a program unit, and certain executable statements  
26 may appear only in certain executable constructs.
- 27 **2.3.1 Executable/Nonexecutable Statements.** Program execution is a sequence, in time, of  
28 computational actions. An executable statement is an instruction to perform or control one or  
29 more of these actions. Thus, the executable statements of a program unit determine the computa-  
30 tional behavior of the program unit. The executable statements are all of those that make up the  
31 syntactic class of *executable-construct*.
- 32 Nonexecutable statements do not specify actions; they are used to configure the program environ-  
33 nent in which computational actions take place. The nonexecutable statements are all those not  
34 classified as executable. All statements in a block data program unit must be nonexecutable. A  
35 module may contain executable statements only within a subprogram in the module.
- 36 **2.3.2 Statement Order.** The syntax rules of Section 2.1 specify the statement order within pro-  
37 gram units and subprograms. These rules are illustrated in Figure 2.1 and Table 2.1. Figure 2.1  
38 shows the ordering rules for statements and applies to all program units and subprograms. Verti-  
39 cal lines delineate varieties of statements that may be interspersed and horizontal lines delineate  
40 varieties of statements that must not be interspersed. USE statements, if any, must appear imme-  
41 diately after the program unit heading. Internal or module subprograms must follow a CON-  
42 TAINS statement. Between USE and CONTAINS statements in a subprogram, nonexecutable  
43 statements generally precede executable statements, though the FORMAT statement, DATA state-  
44 ment, and ENTRY statement may appear among the executable statements. Table 2.1 shows which  
45 statements are allowed in a scoping unit.

1 **Figure 2.1** Requirements on Statement Ordering

PROGRAM, FUNCTION, SUBROUTINE, MODULE, or BLOCK DATA Statement		
USE Statements		
FORMAT and ENTRY Statements	IMPLICIT NONE	
	PARAMETER Statements	IMPLICIT Statements
	PARAMETER and DATA Statements	Derived-Type Definitions, Interface Blocks, Type Declaration Statements, Statement Function Statements, and Specification Statements
	DATA Statements	Executable Constructs
CONTAINS Statement		
Internal Subprograms or Module Subprograms		
END Statement		

32 **Table 2.1** Statements Allowed in Scoping Units

Kind of Scoping Unit:	Main Program	Module	Block Data	External Subprog	Module Subprog	Internal Subprog	Interface Body
USE Statement	Yes	Yes	Yes	Yes	Yes	Yes	Yes
ENTRY Statement	No	No	No	Yes	Yes	No	No
FORMAT Statement	Yes	No	No	Yes	Yes	Yes	No
Misc. Declarations (See Note)	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Derived-Type Definition	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Interface Block	Yes	Yes	No	Yes	Yes	Yes	Yes
Statement Function	Yes	No	No	Yes	Yes	Yes	No
Executable Statement	Yes	No	No	Yes	Yes	Yes	No
CONTAINS	Yes	Yes	No	Yes	Yes	No	No

44 Note: Misc. Declarations are PARAMETER Statements, IMPLICIT Statements, DATA Statements,  
 45 Type Declaration Statements, and Specification Statements. Note that the scoping unit of a module  
 46 does not include any module subprograms that the module contains.

47 **2.3.3 The END Statement.** An *end-program-stmt*, *end-function-stmt*, *end-subroutine-stmt*, *end-*  
 48 *module-stmt*, or *end-block-data-stmt* is an END statement. Each program unit and internal subpro-  
 49 gram must have exactly one END statement, which may be labeled; it must be the last statement of  
 50 the program unit or internal subprogram. The *end-program-stmt*, *end-function-stmt*, and *end-*  
 51 *subroutine-stmt* statements are executable, and may be branch target statements. Executing an *end-*  
 52 *program-stmt* causes termination of execution of the executable program. Executing an *end-*  
 53 *function-stmt* or *end-subroutine-stmt* is equivalent to executing a *return-stmt* in a subprogram.

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

2 **2.3.4 Execution Sequence.** Execution of an executable program begins with the first executable  
 3 construct of the main program. The execution of a main program or subprogram involves execu-  
 4 tion of the executable constructs within its scoping unit. When a procedure is invoked, execution  
 5 begins with the first executable construct appearing after the invoked entry point. With the fol-  
 6 lowing exceptions, the effect of execution is as if the executable constructs are executed in the  
 7 order in which they appear in the main program or subprogram until a STOP, RETURN, or END  
 8 statement is executed. The exceptions are:

9 (1) Execution of a branching statement (8.2) changes the execution sequence. These state-  
 10 ments explicitly specify a new starting place for the execution sequence.

11 (2) IF constructs, CASE constructs, and DO constructs contain an internal statement struc-  
 12 ture and execution of these constructs involves implicit (i.e., automatic) internal branch-  
 13 ing. See Section 8 for the detailed semantics of each of these constructs.

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

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

17 **2.4 Data Concepts.** Nonexecutable statements are used to define the characteristics of the  
 18 data environment. This includes typing variables, declaring arrays, and defining new data types.

19 **2.4.1 Data Type.** A data type is a named category of data that is characterized by a set of values,  
 20 together with a way to denote these values and a collection of operations that interpret and manip-  
 21 ulate the values. This central concept is described in 4.1.

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

23 **2.4.1.1 Intrinsic Type.** An intrinsic type is a type that is defined implicitly, along with opera-  
 24 tions, and is always accessible. The intrinsic types are INTEGER, REAL, COMPLEX, CHARAC-  
 25 TER, and LOGICAL. The properties of intrinsic types are described in 4.3. An intrinsic type may  
 26 be parameterized, in which case the set of data values depends on the values of the parameters.  
 27 Such a parameter is called a **type parameter** (4.3). The type parameters are KIND and LEN.

28 The kind type parameter indicates the decimal range for the integer type (4.3.1.1), the decimal pre-  
 29 cision and exponent range for the real and complex types (4.3.1.2, 4.3.1.3), and the representation  
 30 methods for the character and logical types (4.3.2.1, 4.3.2.2).

*The length type parameter specifies the number of characters for the character type.*

31 **2.4.1.2 Derived Type.** A derived type is a type that is not defined implicitly but requires a type  
 32 definition to declare components of intrinsic or of other derived types. A scalar object of such a  
 33 derived type is called a **structure** (5.1.1.7). The only intrinsic operation for derived types is assign-  
 34 ment with type agreement (4.4.5). For each derived type, structure constructors are available to  
 35 provide values (4.4.4). In addition, data objects of derived type may be used as procedure argu-  
 36 ments and function results, and may appear in input/output lists. If additional operations are  
 37 needed for a derived type, they must be supplied as procedure definitions.

38 Derived types are described further in 4.4.

39 **2.4.2 Data Value.** Each intrinsic type has associated with it a set of values that a datum of that  
 40 type may take. The values for each intrinsic type are described in 4.3. Because derived types are  
 41 ultimately specified in terms of components of intrinsic types, the values that objects of a derived  
 42 type may assume are determined by the type definition and the sets of values of the intrinsic types.

43 **2.4.3 Data Entity.** A data entity is a data object, the result of the evaluation of an expression, or  
 44 the result of the execution of a function reference (called the function result). A data entity has a  
 45 data type (either intrinsic or derived) and has, or may have, a data value (the exception is an unde-  
 46 fined variable). Every data entity has a rank and is thus either a scalar or an array.

1 **2.4.3.1 Data Object.** A data object (often abbreviated to object) is a constant (4.1.2), a variable  
 2 (6), or a subobject of a constant. The type of a named data object may be specified explicitly (5) or  
 3 implicitly (5.3).

4 **Subobjects** are portions of certain named objects that may be referenced and defined (variables  
 5 only) independently of the other portions. These include portions of arrays (array elements and  
 6 array sections), portions of character strings (substrings), and portions of structures (components).  
 7 Subobjects are themselves data objects, but subobjects are referenced only by subobject designa-  
 8 tors. A subobject of a variable is a variable. Subobjects are described in Section 6.

9 Objects referenced by a name are:

- 10 a named scalar (a scalar object)
- 11 a named array (an array object)

12 Subobjects referenced by a subobject designator are:

- 13 an array element (a scalar subobject)
- 14 an array section (an array subobject)
- 15 a structure component (a scalar or an array subobject)
- 16 a substring (a scalar subobject)

17 **2.4.3.1.1 Variable.** A variable <sup>may be</sup> has a value ~~that can become~~ defined and redefined during execu-  
 18 tion of an executable program.

19 **2.4.3.1.2 Constant.** A constant has a value <sup>and</sup> that cannot become defined or redefined during execu-  
 20 tion of an executable program. A constant with a name is called a **named constant** and has the  
 21 PARAMETER attribute (5.1.2.1). A constant without a name is called a **literal constant** (4.3).

22 **2.4.3.1.3 Constant Subobject.** A constant subobject <sup>be</sup> has a value ~~that cannot become defined or~~  
 23 redefined during execution of an executable program, <sup>but</sup> that may depend on the value of a vari-  
 24 able. For example, given: <sub>its value</sub>

```
25 CHARACTER (LEN = 10), PARAMETER :: DIGITS = '0123456789'
26 CHARACTER (LEN = 1)           :: DIGIT
27 ...
28 DIGIT = DIGITS (I:I)
```

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

30 **2.4.3.2 Expression.** An expression produces a data entity when evaluated. An expression (7.1)  
 31 represents either a data reference or a computation, and is formed from operands, operators, and  
 32 parentheses. The type, value, and rank of an expression result are determined by the rules in Sec-  
 33 tion 7.

34 **2.4.3.3 Function Reference.** A function reference (12.4.2) produces a data entity when the func-  
 35 tion is executed during expression evaluation ~~or by a defined operation~~. The type and rank of a  
 36 function result are determined by the interface of the function (12.2.2). The value of a function <sup>gone?</sup>  
 37 result is determined by execution of the function.

38 **2.4.4 Scalar.** A scalar is a datum that is not an array. Scalars may be of any intrinsic type or  
 39 derived type. Note that a structure is scalar even if it has arrays as components. The rank of a sca-  
 40 lar is zero. The shape of a scalar is represented by a rank-one array of size zero.

41 **2.4.5 Array.** An array is a set of scalar data, all of the same type and type parameters, whose indi-  
 42 vidual elements are arranged in a rectangular pattern. An array element is one of the individual  
 43 elements in the array and is a scalar. An array section is a subset of the elements of an array and is  
 44 itself an array.

45 An array with a name has one subscript for each dimension of the pattern. The pattern may have  
 46 up to seven dimensions, and any extent (size) in any dimension. The rank of the array is the

1 number of dimensions, and its size is the total number of elements which is equal to the product of  
2 the extents. An array may have zero size. The shape of an array is determined by its rank and its  
3 extent in each dimension, and may be represented as a rank-one array whose elements are the  
4 extents. All named arrays must be declared, and the rank of a named array is specified in its decla-  
5 ration. The rank of a named array, once declared, is constant and the extents may be constant also.  
6 However, the extents may vary during execution for a dummy argument array, an automatic  
7 array, a pointer array, and an allocatable array.

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

15 <sup>rank one</sup> An array may be constructed from scalars and other ~~rank-one~~ arrays and may be reshaped into  
16 any allowable array shape (4.5).

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

18 **2.4.6 Pointer.** A pointer is a variable that has the POINTER attribute. ~~It must not be referenced~~  
19 ~~or defined until it is associated with a target by allocation (6.3.1) or pointer assignment (7.5.2).~~ A  
20 pointer is disassociated following execution of a NULLIFY or DEALLOCATE statement or follow-  
21 ing pointer association with a disassociated pointer. A disassociated pointer is not currently asso-  
22 ciated with a target (14.6.2). If it is an array, the rank is declared, but the extents are determined  
23 when the pointer is associated with a target. *A pointer is associated with a target*  
*by allocation (6.3.1) or pointer assignment (7.5.2). A pointer must not*  
*be referenced or defined until it is associated.*

24 **2.4.7 Storage.** Many of the facilities of this standard make no assumptions about the physical  
25 storage characteristics of data objects. However, program units that include storage association  
26 dependent features must observe certain storage constraints (14.6.3).

27 **2.5 Fundamental Terms.** The following terms are defined here and used throughout this  
28 standard.

29 **2.5.1 Name and Designator.** A name is used to identify a program constituent, such as a pro-  
30 gram unit, named variable, named constant, dummy argument, or derived type. The rules gov-  
31 erning the construction of names are given in 3.2.2. A **subobject designator** is a name followed by  
32 one or more of the following: component selectors, array section selectors, array element selectors,  
33 and substring selectors.

34 **2.5.2 Keyword.** The term **keyword** is used in two ways in this standard. A word that is part of  
35 the syntax of a statement is a **statement keyword**. These keywords are not reserved words; that is,  
36 names with the same spellings are allowed. Examples of statement keywords are: IF, READ,  
37 UNIT, KIND, and INTEGER.

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

41 **2.5.3 Declaration.** The term **declaration** refers to the specification of attributes for various pro-  
42 gram entities. Often this involves specifying the data type of a named data object or specifying the  
43 shape of a named array object.

44 **2.5.4 Definition.** The term **definition** is used in two ways. First, when a data object is given a  
45 valid value during program execution, it is said to become **defined**. This is often accomplished by  
46 execution of an assignment statement or input statement. Under certain circumstances, a variable  
47 does not have a predictable value and is said to be **undefined**. Section 14 describes the ways in

*no also*



1 which variables may become defined and undefined. The second use of the term **definition** refers  
2 to the declaration of derived types and procedures.

3 **2.5.5 Reference.** A **data object reference** is the appearance of the data object name or data sub-  
4 object designator in a context requiring its value at that point during execution.

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

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

10 A **module reference** is the appearance of a module name in a USE statement (11.3.1).

11 **2.5.6 Association.** **Association** may be name association (14.6.1), pointer association (14.6.2), or  
12 storage association (14.6.3). Name association may be argument association, host association, or  
13 use association.

14 Storage association causes different entities to have related values due to using the same storage.  
15 Any association permits an entity to be identified by different names in the same scoping unit or  
16 by the same name or different names in different scoping units.

17 **2.5.7 Intrinsic.** The qualifier **intrinsic** signifies that the term to which it is applied is defined in  
18 this standard. Intrinsic applies to data types, procedures, and operators. All intrinsic data types,  
19 procedures, and operators may be used in any scoping unit without further definition or specifica-  
20 tion.

21 **2.5.8 Operator.** An **operator** specifies a particular computation involving one (unary operator)  
22 or two (binary operator) data values (operands). Fortran contains a number of intrinsic operators  
23 (e.g., the arithmetic operators +, -, \*, /, and \*\* with numeric operands and the logical operators  
24 .AND., .OR., etc. with logical operands). Additional operators also may be defined within an exe-  
25 cutable program (7.1.3).

26 **2.5.9 Sequence.** A **sequence** is a set ordered by a one-to-one correspondence with the numbers  
27 1, 2, ~~through~~  $n$ . The number of elements in the sequence is  $n$ . A sequence may be empty, in which  
28 case it contains no elements.

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



### 3. CHARACTERS, LEXICAL TOKENS, AND SOURCE FORM

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

3 **3.1 Processor Character Set.** The processor character set is processor dependent. The struc-  
4 ture of a processor character set is:

- 5 (1) **Control characters** ('newline', for example)
- 6 (2) **Graphic characters**
  - 7 (a) Letters (3.1.1)
  - 8 (b) Digits (3.1.2)
  - 9 (c) Underscore (3.1.3)
  - 10 (d) Special characters (3.1.4)
  - 11 (e) Other characters (3.1.5)

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

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

15 R302 *alphanumeric-character* is *letter*  
16 or *digit*  
17 or *underscore*

18 Except for the currency symbol, the graphics used for the characters must be as given in 3.1.1, 3.1.2,  
19 3.1.3, and 3.1.4. However, the style of any graphic is not specified.

20 **3.1.1 Letters.** The twenty-six letters are:

21 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

22 The set of letters define the syntactic class *letter*.

23 If a processor also permits lower-case letters, the lower-case letters are equivalent to the corre-  
24 sponding upper-case letters in program units except in a character context (3.3). However, the  
25 interpretation of case is processor dependent in the FILE= specifier (9.3.4.1, 9.6.1.1) and the  
26 NAME= specifier (9.6.1.6).

*still allows a processor to be case-sensitive in I/O control strings.*

27 **3.1.2 Digits.** The ten digits are:

28 0 1 2 3 4 5 6 7 8 9

29 The ten digits define the syntactic class *digit*. The digits are interpreted according to the number  
30 system of the constant in which they are used: binary, octal, decimal, or hexadecimal.

31 **3.1.3 Underscore.**

32 R303 *underscore* is \_

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

34 **3.1.4 Special Characters.** The twenty-one special characters are *shown in table 3.1*

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

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

21 **3.1.5 Other Characters.** Additional characters may be representable in the processor, but may  
 22 appear only in character constants, character string edit descriptors, comments, and input/output  
 23 records (4.3.2.1, 10.2.1, 3.3.1.1, 3.3.2.1, 9.1.1).

24 The default character type must support a character set that includes the Fortran character set.  
 25 Other character sets may be supported by the processor in terms of nondefault character types.  
 26 The characters available in the nondefault character types are not specified, except that one charac-  
 27 ter in each nondefault character type must be designated as a blank character to be used as a pad-  
 28 ding character.

29 **3.2 Low-Level Syntax.** The low-level syntax describes the fundamental lexical tokens of a pro-  
 30 gram unit. Lexical tokens are sequences of characters with indivisible interpretations that consti-  
 31 tute the building blocks of a program. They are keywords, names, literal constants other than  
 32 complex literal constants, operators, labels, delimiters, comma, =, =>, :, ::, ;, and %.

33 **3.2.1 Keywords.** Keywords appear as upper-case words in the syntax rules in Sections 4 through  
 34 12.

35 **3.2.2 Names.** Names are used for various entities such as variables, program units, dummy  
 36 arguments, named constants, and derived types.

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

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

39 Examples of names:

40	A1	
41	NAME_LENGTH	(single underscore)
42	S_P_R_E_A_D__O_U_T	(two consecutive underscores)
43	TRAILER_	(trailing underscore)

44 **3.2.3 Constants.**

45 R305 *constant* is *literal-constant*  
 46 or *named-constant*

47 R306 *literal-constant* is *int-literal-constant*  
 48 or *real-literal-constant*  
 49 or *complex-literal-constant*

1		or <i>logical-literal-constant</i>
2		or <i>char-literal-constant</i>
3		or <i>boz-literal-constant</i>
4	R307 <i>named-constant</i>	is <i>name</i>
5	R308 <i>int-constant</i>	is <i>constant</i>
6	Constraint: <i>int-constant</i> must be of type integer.	
7	R309 <i>char-constant</i>	is <i>constant</i>
8	Constraint: <i>char-constant</i> must be of type character.	
9	<b>3.2.4 Operators.</b>	
10	R310 <i>intrinsic-operator</i>	is <i>power-op</i>
11		or <i>mult-op</i>
12		or <i>add-op</i>
13		or <i>concat-op</i>
14		or <i>rel-op</i>
15		or <i>not-op</i>
16		or <i>and-op</i>
17		or <i>or-op</i>
18		or <i>equiv-op</i>
19	R708 <i>power-op</i>	is **
20	R709 <i>mult-op</i>	is *
21		or /
22	R710 <i>add-op</i>	is +
23		or -
24	R712 <i>concat-op</i>	is //
25	R714 <i>rel-op</i>	is .EQ.
26		or .NE.
27		or .LT.
28		or .LE.
29		or .GT.
30		or .GE.
31		or ==
32		or /=
33		or <
34		or <=
35		or >
36		or >=
37	R719 <i>not-op</i>	is .NOT.
38	R720 <i>and-op</i>	is .AND.
39	R721 <i>or-op</i>	is .OR.
40	R722 <i>equiv-op</i>	is .EQV.
41		or .NEQV.
42	R311 <i>defined-operator</i>	is <i>defined-unary-op</i>
43		or <i>defined-binary-op</i>
44		or <i>extended-intrinsic-op</i>
45	R704 <i>defined-unary-op</i>	is . letter [ letter ] ... .
46	R724 <i>defined-binary-op</i>	is . letter [ letter ] ... .
47	R312 <i>extended-intrinsic-op</i>	is <i>intrinsic-operator</i>



- 1 One or more blanks must be used to separate certain adjacent keywords and may be optionally  
 2 used between others, as follows.

Blanks Optional	Blank Mandatory
BLOCK DATA	CASE DEFAULT
DOUBLE PRECISION	IMPLICIT <i>type-spec</i>
ELSE IF	IMPLICIT NONE <del>MODULE</del> <del>PROCEDURE</del>
END BLOCK DATA	RECURSIVE FUNCTION
END DO	RECURSIVE SUBROUTINE
END FILE	RECURSIVE <i>type-spec</i>
END FUNCTION	<i>type-spec</i> FUNCTION
END IF	<i>type-spec</i> RECURSIVE
END INTERFACE	
END MODULE	
END PROGRAM	
END SELECT	
END SUBROUTINE	
END TYPE	
END WHERE	
GO TO	
IN OUT	
SELECT CASE	

22 **3.3.1.1 Free Form Commentary.** The character “!” initiates a comment except when it appears  
 23 within a character context. The comment extends to the end of the source line. If the first non-  
 24 blank character on a line is an “!”, the line is called a comment line. Lines containing only blanks  
 25 or containing no characters are also comment lines. Comments may appear anywhere in a pro-  
 26 gram unit and may precede the first statement of a program unit. Comments have no effect on the  
 27 interpretation of the program unit.

28 **3.3.1.2 Free Form Statement Separation.** The character “;” separates statements, or partial  
 29 statements, on a single source line except when it appears in a character context or in a comment.  
 30 If a “;” separator is followed by zero or more blanks and one or more “;” separators, the sequence  
 31 from the first “;” to the last, inclusive, is interpreted as a single “;” separator.

32 **3.3.1.3 Free Form Statement Continuation.** The character “&” is used to indicate that the cur-  
 33 rent statement is continued on the next line that is not a comment line. Comment lines cannot be  
 34 continued; an “&” in a comment has no effect. Comments may occur within a continued state-  
 35 ment. When used for continuation, the “&” is not part of the statement. No line may contain a  
 36 single “&” as the only nonblank character or as the only nonblank character before an “!”.

37 **3.3.1.3.1 Noncharacter Context Continuation.** If an “&” not in a comment is the last nonblank  
 38 character on a line or the last nonblank character before an “!”, the statement is continued on the  
 39 next line that is not a comment line. If the first nonblank character on the next noncomment line is  
 40 an “&”, the statement continues at the next character position following the “&”; otherwise, it con-  
 41 tinues with the first character position of the next noncomment line.

42 If a lexical token is split across the end of a line, the first nonblank character on the first following  
 43 noncomment line must be an “&” immediately followed by the successive characters of the split  
 44 token.

45 **3.3.1.3.2 Character Context Continuation.** If a character context is to be continued, the “&”  
 46 must be the last nonblank character on the line and must not be followed by commentary. An “&”  
 47 must be the first nonblank character on the next line that is not a comment line and the statement  
 48 continues with the next character following the “&”.

## Characters, Lexical Tokens, and Source Form

1 **3.3.1.4 Free Form Statements.** A label may precede any statement not forming part of another  
2 statement. Note that no Fortran statement begins with a digit. A free form statement must not  
3 have more than 39 continuation lines.

4 **3.3.2 Fixed Source Form.** In fixed source form, there are restrictions on where a statement may  
5 appear within a line. If a source line contains only default kind characters, it must contain exactly  
6 72 characters; otherwise, its maximum number of characters is processor dependent.

7 Except in a character context, blanks are insignificant and may be used freely throughout the pro-  
8 gram.

9 **3.3.2.1 Fixed Form Commentary.** The character "!" initiates a comment except when it appears  
10 within a character context or in character position 6. The comment extends to the end of the line.  
11 If the first nonblank character on a line is an "!" in any character position other than character  
12 position 6, the line is a comment line. Lines beginning with a "C" or "\*" in character position 1  
13 and lines containing only blanks are also comments. Comments may appear anywhere within a  
14 program unit and may precede the first statement of the program unit. Comments have no effect  
15 on the interpretation of the program unit.

16 **3.3.2.2 Fixed Form Statement Separation.** The character ";" separates statements, or partial  
17 statements, on a single source line except when it appears in a character context or in a comment.  
18 If a ";" separator is followed by zero or more blanks and one or more ";" separators, the sequence  
19 from the first ";" to the last, inclusive, is interpreted as a single ";" separator.

20 **3.3.2.3 Fixed Form Statement Continuation.** Except within commentary, character position 6 is  
21 used to indicate continuation. If character position 6 contains a blank or zero, this line is the initial  
22 line of a new statement which begins in character position 7. If character position 6 contains any  
23 character other than blank or zero, character positions 7-72 of this line constitute a continuation of  
24 the preceding noncomment line. Note that an "!" or ";" in character position 6 indicates a continu-  
25 ation of the preceding noncomment line. Comment lines cannot be continued. Comment lines  
26 may occur within a continued statement.

27 **3.3.2.4 Fixed Form Statements.** A label, if present, must occur in character positions 1 through 5  
28 of the first line of a statement; otherwise, positions 1 through 5 must be blank. Blanks may appear  
29 anywhere within a label. Note that a statement following a ";" on the same line must not be  
30 labeled. Character positions 1 through 5 of any continuation lines must be blank. A fixed form  
31 statement must not have more than 19 continuation lines. The program unit END statement must  
32 not be continued and no other statement in the program unit may have an initial line that appears  
33 to be a program unit END statement.

34 **3.4 Including Source Text.** Additional text may be incorporated into the source text of a pro-  
35 gram unit during processing. This is accomplished with the INCLUDE line, which has the form

36 INCLUDE *char-literal-constant*

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

38 An INCLUDE line is not a Fortran statement.

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

42 The effect of the INCLUDE line is as if the referenced source text physically replaced the  
43 INCLUDE line prior to program processing. Included text may contain any source text, including  
44 additional INCLUDE lines; such nested INCLUDE lines are similarly replaced with the specified  
45 source text. The maximum depth of nesting of any nested INCLUDE lines is processor dependent.

46 An INCLUDE line must not reference itself directly or indirectly.



- 1 When an INCLUDE line is resolved, the first included statement line must not be a continuation
- 2 line and the last included statement line must not be continued.
- 3 The interpretation of *char-literal-constant* is processor dependent. An example of a possible valid
- 4 interpretation is that *char-literal-constant* is the name of a file that contains the source text to be
- 5 included.



## 4. INTRINSIC AND DERIVED DATA TYPES

1 Fortran provides an abstract means whereby data may be categorized without relying on a partic-  
2 ular physical representation. This abstract means is the concept of **data type**. Each data type has a  
3 name. The names of the intrinsic types are predefined by the language; the names of any derived  
4 types must be defined in type definitions (4.4.1). A data type is characterized by a set of values, a  
5 means to denote the values, and a set of operations that can manipulate and interpret the values.

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

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

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

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

20 In addition to the intrinsic types, application specific types may be derived. Objects of derived  
21 type have **components**. Each component is of an intrinsic type or of a derived type. A type defini-  
22 tion (4.4.1) is required to supply the name of the type and the names and types of its components.  
23 For example, if the complex type were not intrinsic but had to be derived, a type definition would  
24 be required to supply the name "complex" and declare two components, each of type real.

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

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

30 **4.1 The Concept of Data Type.** A data type has (1) a name, (2) a set of valid values, (3) a  
31 means to denote such values (constants), and (4) a set of operations to manipulate the values.

32 **4.1.1 Set of Values.** For each data type, there is a set of valid values. The set of valid values may  
33 be completely determined, as is the case for logical, or may be determined by a processor-  
34 dependent method, as is the case for integer and real. For complex or derived types, the set of  
35 valid values consists of the set of all the combinations of the values of the individual components.  
36 For parameterized types, the set of valid values depends on the values of the parameters.

37 **4.1.2 Constants.** For each of the intrinsic data types, the syntax for literal constants of that type is  
38 specified in this standard. These literal constants are described in 4.3 for each intrinsic type.  
39 Within an executable program, all literal constants that have the same form have the same value.

40 A constant value may be given a name (5.1.2.1, 5.2.10).

41 A constant value of derived type may be constructed (4.4.4) using a structure constructor from an  
42 appropriate sequence of constant expressions (7.1.6.1). Such a constant value is considered to be a  
43 scalar even though the value may have components that are arrays.

44 **4.1.3 Operations.** For each of the intrinsic data types, a set of operations and corresponding  
45 operators are defined intrinsically. These are described in Section 7. The intrinsic set may be aug-  
46 mented with operations and operators defined by functions with the OPERATOR interface  
47 (12.3.2.1). Operator definitions are described in Sections 7 and 12.

## Intrinsic and Derived Data Types

- 1 For derived types, the only intrinsic operation is assignment. All other operations must be defined
- 2 by the executable program (4.4.5).

3 **4.2 Relationship of Types and Values to Objects.** The name of a data type serves as a  
4 type specifier and may be used to declare objects of that type. A declaration specifies the type of a  
5 named object. A data object may be declared explicitly or implicitly. Once a derived type is  
6 defined, an object may be declared to be of that type. Data objects may have attributes in addition  
7 to their types. Section 5 describes the way in which a data object is declared and how its type and  
8 other attributes are specified.

9 An array is a collection of subobjects.

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

13 A scalar object of derived type is referred to as a structure. The components of a structure are sub-  
14 objects.

15 Variables may be objects or subobjects. The data type of a variable determines which values that  
16 variable may take. Assignment provides one means of defining or redefining the value of a vari-  
17 able of any type. Assignment is defined intrinsically for all types when the type, type parameters,  
18 and shape of both the variable and the value to be assigned to it are identical. Assignment between  
19 objects of certain differing intrinsic types, type parameters, and shapes is described in Section 7.  
20 For example, assignment of an integer value to a real variable is defined intrinsically. A subrou-  
21 tine (7.5.1.3) with the ASSIGNMENT interface (12.3.2.1) defines an assignment that is not defined  
22 intrinsically or redefines an intrinsic derived-type assignment.

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

24 **4.3 Intrinsic Data Types.** The intrinsic data types are:

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

27 **4.3.1 Numeric Types.** The numeric types are provided for numerical computation. The normal  
28 operations of arithmetic, addition (+), subtraction (-), multiplication (\*), division (/), exponentia-  
29 tion (\*\*), negation (unary -), and identity (unary +), are defined intrinsically for this set of types.

30 Each numeric type includes a zero value, which is considered to be neither negative nor positive.  
31 The value of a signed zero is the same as the value of an unsigned zero. In this standard, the  
32 unqualified term "literal constant" means "unsigned literal constant" when applied to numeric  
33 types.

34 **4.3.1.1 Integer Type.** The set of values for the integer type is a subset of the mathematical inte-  
35 gers. A processor must provide one or more representation methods that define sets of values for  
36 data of type integer. Each such method is characterized by a value for a type parameter called the  
37 kind type parameter. The kind type parameter of a representation method is returned by the  
38 intrinsic inquiry function KIND (13.13.51). Among the kind values that provide a given range, one  
39 providing the smallest range is returned by the intrinsic function SELECTED\_INT\_KIND  
40 (13.13.92). The decimal exponent range of a representation method is returned by the intrinsic  
41 function RANGE (13.13.85).

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

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

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

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

- 1 R402 *digit-string* is *digit* [ *digit* ] ...
- 2 R403 *signed-int-literal-constant* is [ *sign* ] *int-literal-constant*
- 3 R404 *int-literal-constant* is *digit-string* [ *\_ kind-param* ]
- 4 R405 *kind-param* is *digit-string*
- 5 or *scalar-int-constant-name*
- 6 R406 *sign* is +
- 7 or -

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

9 Constraint: The value of *kind-param* must specify a representation method that exists on the processor.

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

13 Examples of unsigned and signed integer literal constants are:

- 14 473
- 15 +56
- 16 -101
- 17 21\_2
- 18 21\_SHORT
- 19 1976354279568241\_8

20 where SHORT is a scalar ~~nonnegative~~ integer named constant whose value must be a valid kind type parameter value for the integer type.

22 An integer constant is interpreted as a decimal value.

23 In a DATA statement (5.2.9), an unsigned binary, octal, or hexadecimal literal constant must correspond to an integer scalar ~~entity~~ *variable*.

- 25 R407 *boz-literal-constant* is *binary-constant*
- 26 or *octal-constant*
- 27 or *hex-constant*

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

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

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

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

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

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

- 37 R411 *hex-digit* is *digit*
- 38 or A
- 39 or B
- 40 or C
- 41 or D
- 42 or E
- 43 or F

44 In these constants, the binary, octal, and hexadecimal digits are interpreted according to their respective number systems. *h.c. OK & equivalent*

## Intrinsic and Derived Data Types

1 **4.3.1.2 Real Type.** The real type has values that approximate the mathematical real numbers. A  
2 processor must provide two or more approximation methods that define sets of values for data of  
3 type real. Each such method has a representation method and is characterized by a value for a  
4 type parameter called the kind type parameter. The kind type parameter of an approximation  
5 method is returned by the intrinsic inquiry function KIND (13.13.51). Among the kind values that  
6 provide a given precision and a given exponent range, one providing the smallest decimal preci-  
7 sion is returned by the intrinsic function SELECTED\_REAL\_KIND (13.13.93). The decimal preci-  
8 sion and decimal exponent range of an approximation method are returned by the intrinsic func-  
9 tions PRECISION (13.13.79) and RANGE (13.13.85).

10 The type specifier for the real type is the keyword REAL and the type specifier for the double pre-  
11 cision real type is the keyword DOUBLE PRECISION.

12 If the type keyword REAL is specified and the kind type parameter is not specified, the default  
13 kind value is KIND (0.0) and the data entity is of type default real. If the type keyword DOUBLE  
14 PRECISION is specified, a kind type parameter must not be specified and the data entity is of type  
15 double precision real. The kind type parameter of such an entity has the value KIND (0.0D0).  
16 The decimal precision of the double precision real approximation method must be greater than  
17 that of the default real method.

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

19 R413 *real-literal-constant* is *significand* [ *exponent-letter exponent* ] [ *\_kind-param* ]  
20 or *digit-string exponent-letter exponent* [ *\_kind-param* ]

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

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

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

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

27 Constraint: The value of *kind-param* must specify an approximation method that exists on the  
28 processor.

29 A real literal constant without a kind type parameter is a default real constant if it is without an  
30 exponent part or has exponent letter E, and is a double precision real constant if it has exponent  
31 letter D. A real literal constant written with a kind type parameter is a real constant with the speci-  
32 fied kind type parameter.

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

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

37 Examples of signed real literal constants are:

38 -12.78

39 +1.6E3

40 2.1

41 -16.E4\_8

42 0.45E-4

43 10.93E7\_QUAD

44 .123

45 3E4

46 In 10.93E7\_QUAD, the <sup>signed</sup> ~~named~~ integer constant QUAD must have been defined and its value must  
47 be a kind type parameter value for the real type.

↑  
valid

- 1 **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.
- 2  
3
- 4 Each approximation method used to represent data entities of type real is available for both the real and imaginary parts of a data entity of type complex. A kind type parameter may be specified for a complex entity and selects for both parts the real approximation method characterized by this kind type parameter value.
- 5  
6  
7
- 8 If a kind type parameter is not specified, the type of both parts is default real and the complex data entity is default complex.
- 9
- 10 The type specifier for the complex type is the keyword COMPLEX. There is no keyword for double precision complex.
- 11
- 12 R417 *complex-literal-constant* is ( *real-part* , *imag-part* )
- 13 R418 *real-part* is *signed-int-literal-constant*
- 14 or *signed-real-literal-constant*
- 15 R419 *imag-part* is *signed-int-literal-constant*
- 16 or *signed-real-literal-constant*
- 17 If the real part and imaginary part of a complex literal constant are both real but do not have the same kind type parameter values, the part with the lesser decimal precision or, if these are the same, the part with the lesser parameter value is converted to the approximation method of the other part. The kind type parameter value of the complex literal constant is the parameter value of the part with the greater decimal precision or greater parameter value if the decimal precisions are the same.
- 18  
19  
20  
21  
22
- 23 If both the real and imaginary parts are signed integer literal constants, they are converted to the default real approximation method and the constant is of type default complex. If only one of the parts is a signed integer literal constant, the signed integer literal constant is converted to the approximation method selected for the signed real literal constant and the kind type parameter value of the complex literal constant is that of the signed real literal constant.
- 24  
25  
26  
27
- 28 Examples of complex literal constants are:
- 29 (1.0, -1.0)
- 30 (3, 3.1E6)
- 31 (4.0\_4, 3.6E7\_8)
- 32 **4.3.2 Nonnumeric Types.** The nonnumeric types are provided for nonnumeric processing. The intrinsic operations defined for each of these types are given below.
- 33
- 34 **4.3.2.1 Character Type.** The character type has a set of values composed of character strings. A character string is a sequence of characters, numbered from left to right 1, 2, 3, ... up to the number of characters in the string. The number of characters in the string is called the length of the string. The length is a type parameter; its value is greater than or equal to zero. Strings of different lengths are all of type character.
- 35  
36  
37  
38
- 39 A processor must provide one or more representation methods that define sets of values for data of type character. Each such method is characterized by a value for a type parameter called the kind type parameter. The kind type parameter of a representation method is returned by the intrinsic inquiry function KIND (13.13.51). Any character of a particular representation method representable in the processor may occur in a character string of that representation method.
- 40  
41  
42  
43
- 44 If the kind type parameter is not specified, the default kind value is KIND ('A') and the data entity is of type default character.
- 45
- 46 The type specifier for the character type is the keyword CHARACTER.
- 47 A character literal constant is written as a sequence of characters, delimited by either apostrophes or quotation marks.
- 48

## Intrinsic and Derived Data Types

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

3 Constraint: The value of *kind-param* must specify a representation method that exists on the proc-  
4 essor.

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

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

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

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

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

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

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

22 The intrinsic operation concatenation (//) is defined between two data entities of type character  
23 (7.2.2) with the same kind type parameter.

24 Examples of character literal constants are:

25 "DON' T"  
26 'DON' ' T'

27 both of which have the value DON'T and

28 ''

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

30 Examples of nondefault character literal constants, where the processor supports the correspond-  
31 ing character sets, are:

32 CYRILLIC\_ 'БЕЗ НЕЕ НВУТО НЕ ВОЗМОЖН

33 HANZI\_ '没有她什么部件了'

34 MAGYAR\_ 'Őnélküle, nem csinálhat'

35 NIHONGO\_ '彼女なしでは何もできない'

36 where CYRILLIC, HANZI, MAGYAR, and NIHONGO are named constants whose values are the  
37 kind type parameters for Cyrillic (Russian and other Slavic languages), Hanzi (Chinese), Hungar-  
38 ian, and Hihongo (Japanese) characters, respectively. Note that Hanzi and Hihongo are very large  
39 (ideographic) character sets, whereas Cyrillic and Hungarian are alphabetic.

40 **4.3.2.1.1 Collating Sequence.** Each implementation defines a collating sequence for the charac-  
41 ter set of each kind of character. A collating sequence is a one-to-one mapping of the characters  
42 into the nonnegative integers such that each character corresponds to a different nonnegative inte-  
43 ger. The intrinsic functions CHAR and ICHAR (see Section 13) provide conversions between the  
44 characters and the integers according to this mapping. Thus,

45 ICHAR ('character')

46 returns the integer value of the specified character according to the collating sequence of the proc-  
47 essor.



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

- 2 (1) ICHAR ('A') < ICHAR ('B') < ... < ICHAR ('Z') for the twenty-six letters.
- 3 (2) ICHAR ('0') < ICHAR ('1') < ... < ICHAR ('9') for the ten digits.
- 4 (3) ICHAR (' ') < ICHAR ('0') < ICHAR ('9') < ICHAR ('A') or
- 5 ICHAR (' ') < ICHAR ('A') < ICHAR ('Z') < ICHAR ('0').
- 6 (4) ICHAR ('a') < ICHAR ('b') < ... < ICHAR ('z'), if the processor supports lower-case
- 7 letters.
- 8 (5) ICHAR (' ') < ICHAR ('0') < ICHAR ('9') < ICHAR ('a') or
- 9 ICHAR (' ') < ICHAR ('a') < ICHAR ('z') < ICHAR ('0'), if the processor supports
- 10 lower-case letters.

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

14 ANSI X3.4-1986 (ASCII) assigns numerical codes to a set of characters that includes the letters, dig-  
 15 its, underscore, and special characters; the sequence of such codes is called in this standard the  
 16 **ASCII collating sequence**. The intrinsic functions ACHAR and IACHAR provide conversions  
 17 between these characters and the integers of the ASCII collating sequence. The intrinsic functions  
 18 LGT, LGE, LLE, and LLT provide comparisons between strings based on the ASCII collating  
 19 sequence. ~~Note that X3.4-1986 is the US national version of ISO 646:1977.~~ International portability  
 20 is guaranteed if the set of characters used is limited to the letters, digits, underscore, and special  
 21 characters.

22 **4.3.2.2 Logical Type.** The logical type has two values which represent true and false.

23 A processor must provide one or more **representation methods** for data of type logical. Each such  
 24 method is characterized by a value for a type parameter called the kind type parameter. The kind  
 25 type parameter of a representation method is returned by the intrinsic inquiry function KIND  
 26 (13.13.51).

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

29 R421 *logical-literal-constant* is .TRUE. [ *\_kind-param* ]  
 30 or .FALSE. [ *\_kind-param* ]

31 **Constraint:** The value of *kind-param* must specify a representation method that exists on the proc-  
 32 essor.

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

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

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

41 **4.4 Derived Types.** Additional data types may be derived from the intrinsic data types. A  
 42 type definition is required to define the name of the type and the names and types of its compo-  
 43 nents. Ultimately, a derived type is resolved into components that are either of intrinsic type or are  
 44 pointers. *They are called the ultimate components atomic would be better?*

45 By default, derived types defined in the specification part of a module are accessible (5.1.2.2, 5.2.3)  
 46 in any scoping unit that accesses the module. This default may be changed to restrict the accessi-  
 47 bility of such types to the host module itself. A particular type definition may be declared to be

## Intrinsic and Derived Data Types

- 1 public or private regardless of the default accessibility declared for the module. In addition, a type  
2 may be accessible while its components are private.
- 3 By default, no storage sequence is implied by the order of the component definitions. However, if  
4 the definition of a derived type contains a SEQUENCE statement, the type is a **sequence type**. The  
5 order of the component definitions in a sequence type specifies a storage sequence for objects of  
6 that type.
- 7 The type specifier for derived types is the keyword TYPE followed by the name of the type in  
8 parentheses (R502).

### 9 4.4.1 Derived-Type Definition.

10 R422 *derived-type-def* is *derived-type-stmt*  
11 [ *private-sequence-stmt* ] ...  
12 *component-def-stmt*  
13 [ *component-def-stmt* ] ...  
14 *end-type-stmt*

15 R423 *private-sequence-stmt* is PRIVATE  
16 or SEQUENCE

17 R424 *derived-type-stmt* is TYPE [ [ , *access-spec* ] :: ] *type-name*

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

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

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

24 Constraint: If a component of a derived type is of a type declared to be private, either the derived  
25 type definition must contain the PRIVATE statement or the derived type must be pri-  
26 vate.

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

29 R425 *end-type-stmt* is END TYPE [ <sup>type</sup> *type-name* ]

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

32 R426 *component-def-stmt* is *type-spec* [ [ , *component-attr-spec-list* ] :: ] ■  
33 ■ *component-decl-list*

34 R427 *component-attr-spec* is POINTER  
35 or DIMENSION ( *component-array-spec* )

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

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

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

43 R428 *component-array-spec* is *explicit-shape-spec-list*  
44 or *deferred-shape-spec-list*

45 R429 *component-decl* is *component-name* [ ( *component-array-spec* ) ] ■  
46 ■ [ \* *char-length* ]

- 1 Constraint: If the POINTER attribute is not specified, each *component-array-spec* must be an  
 2 *explicit-shape-spec-list*.
- 3 Constraint: If the POINTER attribute is specified, each *component-array-spec* must be a *deferred-*  
 4 *shape-spec-list*.
- 5 Constraint: The \* *char-length* option is permitted only if the type specified is character.
- 6 Constraint: A *char-length* in a *component-decl* must be a constant specification expression (7.1.6.2).
- 7 Constraint: Each bound in the *explicit-shape-spec* (R428) must be a constant specification expres-  
 8 sion (7.1.6.2).
- 9 If the SEQUENCE statement is present, the type is a sequence type. If all of the ultimate compo-  
 10 nents are of type default integer, default real, double precision real, default complex, or default  
 11 logical and are not pointers, the type is a numeric sequence type. If all of the ultimate components  
 12 are of type default character and are not pointers, the type is a character sequence type.
- 13 Note that the double colon separator in a *component-def-stmt* is required only if the DIMENSION  
 14 attribute, the POINTER attribute, or both are specified; otherwise, it is optional.
- 15 A component is an array if its *component-decl* contains a *component-array-spec* or its *component-def-*  
 16 *stmt* contains the DIMENSION attribute. If the *component-decl* contains a *component-array-spec*, it  
 17 specifies the array bounds (5.1.2.4.1); otherwise, the *component-array-spec* in the DIMENSION attri-  
 18 bute specifies the array bounds.
- 19 The accessibility of a derived type may be declared explicitly by an *access-spec* in its *derived-type-*  
 20 *stmt* or in an *access-stmt*. The accessibility is the default (5.2.3) if it is not declared explicitly. If a  
 21 type definition is PRIVATE, then the type name, the structure value constructor (4.4.4) for the type,  
 22 any entity that is of the type, and any procedure that has a dummy argument or function result  
 23 that is of the type are accessible only within the module containing the definition.
- 24 If a type definition contains a PRIVATE statement, the component names for the type are accessi-  
 25 ble only within the module containing the definition, even if the type itself is public (5.1.2.2). The  
 26 component names and hence the internal structure of the type are inaccessible in any scoping unit  
 27 accessing the module via a USE statement. Similarly, the structure constructor for such a type may  
 28 be employed only within the defining module.
- 29 An example of a derived-type definition is:
- ```
30 TYPE PERSON
31     INTEGER AGE
32     CHARACTER (LEN = 50) NAME
33 END TYPE PERSON
```
- 34 An example of declaring a variable CHAIRMAN of type PERSON is:
- ```
35 TYPE (PERSON) :: CHAIRMAN
```
- 36 A type definition may have a component that is an array. For example:
- ```
37 TYPE LINE
38     REAL, DIMENSION (2, 2) :: COORD      ! X1, Y1, X2, Y2
39     REAL                    :: WIDTH      ! Line width in centimeters
40     INTEGER                  :: PATTERN   ! 1 for solid, 2 for dash, 3 for dot
41 END TYPE LINE
```
- 42 An example of declaring a variable LINE\_SEGMENT to be of the type LINE is:
- ```
43 TYPE (LINE) :: LINE_SEGMENT
```
- 44 The scalar variable LINE\_SEGMENT has a component that is an array. In this case, the array is a  
 45 subobject of a scalar. Note that the double colon in the definition for COORD is required; the dou-  
 46 ble colon in the definition for WIDTH and PATTERN is optional.
- 47 An example of a type with private components is:

## Intrinsic and Derived Data Types

```
1  MODULE DEFINITIONS
2      TYPE POINT
3          PRIVATE
4          REAL :: X, Y
5      END TYPE POINT
6  END MODULE
```

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

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

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

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

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

15 An example of a private type is:

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

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

21 An example of a numeric sequence type is:

```
22 TYPE NUMERIC_SEQ
23     SEQUENCE
24     INTEGER :: INT_VAL
25     REAL    :: REAL_VAL
26     LOGICAL :: LOG_VAL
27 END TYPE NUMERIC_SEQ
```

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

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

34 Any object of type REFERENCE will have the four components VOLUME, YEAR, PAGE, and  
35 TITLE, plus a pointer to an array of characters holding ABSTRACT. The size of this target array  
36 will be determined by the length of the abstract. The space for the target may be allocated (6.3.1)  
37 or the pointer component may be associated with a target in a pointer assignment statement  
38 (7.5.2).

39 A pointer component of a derived type may have as its target an object of the type of which it is a  
40 component. For example:

```
41 TYPE NODE
42     INTEGER                :: VALUE
43     TYPE (NODE), POINTER  :: NEXT_NODE
44 END TYPE
```

45 A type such as this may be used to construct linked lists of objects of type NODE.

1 **4.4.2 Determination of Derived Types.** A particular type name may be defined at most once in a  
 2 scoping unit. Derived-type definitions with the same type name may appear in different scoping  
 3 units, in which case they may be independent and describe different derived types or they may  
 4 describe the same type.

5 Two data entities have the same type if they are declared with reference to the same derived-type  
 6 definition. The definition may be accessed from a module or from a host scoping unit. Data enti-  
 7 ties in different scoping units also have the same type if they are declared with reference to differ-  
 8 ent derived-type definitions that have the same name, have the SEQUENCE property, and have  
 9 structure components that do not have PRIVATE accessibility and agree in order, name, and attri-  
 10 butes. Otherwise, they are of different derived types. A data entity declared using a type with the  
 11 SEQUENCE property is not of the same type as an entity of a type declared to be PRIVATE or  
 12 which has components that are PRIVATE.

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

```
14 TYPE POINT
15     REAL X, Y
16 END TYPE POINT
17 TYPE (POINT) :: X1
18 CALL SUB (X1)
19     ...
20 CONTAINS
21     SUBROUTINE SUB (A)
22         TYPE (POINT) :: A
23         ...
24     END SUBROUTINE SUB
```

25 The definition of derived type POINT is known in subroutine SUB because the scoping unit of SUB  
 26 is contained within the scoping unit of the containing program unit (host association). Because the  
 27 declarations of X1 and A both reference the same derived-type definition, X1 and A have the same  
 28 type. X1 and A also would have the same type if the derived-type definition was in a module and  
 29 both SUB and its containing program unit accessed the module.

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

```
31 PROGRAM PGM
32     TYPE EMPLOYEE
33         SEQUENCE
34         INTEGER          ID_NUMBER
35         CHARACTER (50) NAME
36     END TYPE EMPLOYEE
37     TYPE (EMPLOYEE) PROGRAMMER
38     CALL SUB (PROGRAMMER)
39     ...
40 END PROGRAM PGM

41 SUBROUTINE SUB (POSITION)
42     TYPE EMPLOYEE
43         SEQUENCE
44         INTEGER          ID_NUMBER
45         CHARACTER (50) NAME
46     END TYPE
47     TYPE (EMPLOYEE) POSITION
48     ...
49 END SUBROUTINE SUB
```

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

## Intrinsic and Derived Data Types

1 Suppose the component name `ID_NUMBER` was `ID_NUM` in the subroutine. Because all the com-  
2 ponent names are not identical to the component names in derived type `EMPLOYEE` in the con-  
3 taining program unit, the actual argument `PROGRAMMER` would not be of the same type as the  
4 dummy argument `POSITION`. Thus, the program would not be standard conforming.

5 **4.4.3 Derived-Type Values.** The set of values of a specific derived type consists of all possible  
6 sequences of component values consistent with the definition of that derived type.

7 **4.4.4 Construction of Derived-Type Values.** A derived-type definition implicitly defines a cor-  
8 responding **structure constructor** that allows a scalar value of derived type to be constructed from  
9 a sequence of values, one value for each component of the derived type.

10 R430 *structure-constructor* is *type-name ( expr-list )*

11 The sequence of expressions in a structure constructor specifies component values that must agree  
12 in number, order, and rank with the components of the derived type. If necessary, each value is  
13 converted according to the rules of intrinsic assignment (7.5.1.4) to a value that agrees in type and  
14 type parameters with the corresponding component of the derived type. For nonpointer compo-  
15 nents, the shape of the expression must agree with the shape of the component. A structure con-  
16 structor whose component values are all constant expressions is a derived-type constant expres-  
17 sion. A structure constructor must not appear before the referenced type is defined.

18 This example illustrates a derived-type constant expression using a derived type defined in 4.4.1:

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

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

22 Where a component in the derived type is a pointer, the corresponding constructor expression  
23 must evaluate to an object that would be an allowable target for such a pointer in a pointer assign-  
24 ment statement (7.5.2). For example, if the variable `TEXT` were declared (5.1) to be

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

26 and `BIBLIO` were declared using the derived-type definition `REFERENCE` in 4.4.1

27 `TYPE (REFERENCE) :: BIBLIO`

28 the statement

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

31 is valid and it identifies the `ABSTRACT` component of the object `BIBLIO` with the target object  
32 `TEXT`.

33 Note that a constant expression must not be constructed for a derived type containing a pointer  
34 component because a constant value is not an allowable target in a pointer assignment statement.

35 **4.4.5 Derived-Type Operations and Assignment.** Any operation on derived-type entities or  
36 nonintrinsic assignment for derived-type entities must be defined explicitly by a function or a sub-  
37 routine and a procedure interface block (12.3.2.1). Arguments and function values may be of any  
38 derived or intrinsic type.

39 **4.5 Construction of Array Values.** An array constructor is defined as a sequence of specified  
40 scalar values and is interpreted as a rank-one array whose element values are those specified in the  
41 sequence.

42 R431 *array-constructor* is *( / ac-value-list / )*

43 R432 *ac-value* is *expr*  
44 or *ac-implied-do*

- 1 R433 *ac-implied-do* is (*ac-value-list* , *ac-implied-do-control* )
- 2 R434 *ac-implied-do-control* is *ac-do-variable* = *scalar-int-expr* , ■  
 3 ■ *scalar-int-expr* [ , *scalar-int-expr* ]
- 4 R435 *ac-do-variable* is *scalar-int-variable*
- 5 Constraint: *ac-do-variable* must be a named variable.
- 6 Constraint: Each *ac-value* expression in the ~~sequence~~ *array constructor* must have the same type and type parameters.  
 7
- 8 If an *ac-value* is a scalar expression, its value specifies an element of the array constructor. If an *ac-*  
 9 *value* is an array expression, the values of the elements of the expression, in array element order  
 10 (6.2.2.2), specify the corresponding sequence of elements of the array constructor. If an *ac-value* is  
 11 an *ac-implied-do*, it is expanded to form an *ac-value* sequence under the control of the *ac-do-variable*,  
 12 as in the DO construct (8.1.4.4).
- 13 For an *ac-implied-do*, the loop initialization and execution is the same as for a DO construct. The  
 14 *ac-do-variable* of an *ac-implied-do* that is contained within another *ac-implied-do* must not appear as  
 15 the *ac-do-variable* of the containing *ac-implied-do*.
- 16 An empty sequence forms a zero-sized rank-one array.
- 17 The type and type parameters of an array constructor are those of the *ac-value* expressions.
- 18 If every expression in an array constructor is a constant expression, the array constructor is a con-  
 19 stant expression. An example is:
- 20 REAL X (3)  
 21 X = (/ 3.2, 4.01, 6.5 /)
- 22 A one-dimensional array may be reshaped into any allowable array shape using the RESHAPE  
 23 intrinsic function (13.13.88). An example is:
- 24 Y = RESHAPE (SOURCE = (/ 2.0, (/ 4.5, 4.5 /), X /), SHAPE = (/ 3, 2 /))
- 25 This results in Y having the 3 × 2 array of values:
- 26 2.0 3.2  
 27 4.5 4.01  
 28 4.5 6.5
- 29 Examples of array constructors containing an implied-DO are:
- 30 (/ (I, I = 1, 1075) /)
- 31 and  
 32 (/ 3.6, (3.6 / I, I = 1, N) /)
- 33 Using the type definitions for PERSON and LINE of 4.4.1, an example of the construction of a  
 34 derived-type array value is:
- 35 (/ PERSON (20, 'SMITH'), PERSON (20, 'JONES') /)
- 36 and an example of the construction of a derived-type scalar value with an array component is:
- 37 LINE (RESHAPE ( (/ 0.0, 1.0, 0.0, 2.0 /), (/ 2, 2 /) ), 0.1, 1)
- 38 In the latter example, the RESHAPE intrinsic function is used to construct a value that represents a  
 39 solid line from (0,0) to (1,2) of width 0.1 centimeters.





## 5. DATA OBJECT DECLARATIONS AND SPECIFICATIONS

1 Every data object has a number of properties (for example, type, rank, and shape) that determine  
2 the characteristics of the data and the uses of the object. Collectively, these properties are termed  
3 the **attributes** of the data object. A named data object must not be specified explicitly to have a  
4 particular attribute more than once in a scoping unit. The type of a named data object is either  
5 determined implicitly by the first letter of its name (5.3) or is specified explicitly in a **type declara-**  
6 **tion statement**. Additional attributes also may be specified by separate specification statements;  
7 all of them may be included in a type declaration statement.

8 For example:

9 INTEGER INCOME, EXPENDITURE

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

11 REAL, DIMENSION (-5:+5) :: X, Y, Z

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

### 15 5.1 Type Declaration Statements.

16 R501 *type-declaration-stmt* is *type-spec* [ [ , *attr-spec* ] ... :: ] *entity-decl-list*

17 R502 *type-spec* is INTEGER [ *kind-selector* ]  
18 or REAL [ *kind-selector* ]  
19 or DOUBLE PRECISION  
20 or COMPLEX [ *kind-selector* ]  
21 or CHARACTER [ *char-selector* ]  
22 or LOGICAL [ *kind-selector* ]  
23 or TYPE ( *type-name* )

24 R503 *attr-spec* is PARAMETER  
25 or *access-spec*  
26 or ALLOCATABLE  
27 or DIMENSION ( *array-spec* )  
28 or EXTERNAL  
29 or INTENT ( *intent-spec* )  
30 or INTRINSIC  
31 or OPTIONAL  
32 or POINTER  
33 or SAVE  
34 or TARGET

35 R504 *entity-decl* is *object-name* [ ( *array-spec* ) ] ■  
36 ■ [ \* *char-length* ] [ = *initialization-expr* ]  
37 or *function-name* [ ( *array-spec* ) ] [ \* *char-length* ]

38 R505 *kind-selector* is ( [ KIND = ] *scalar-int-initialization-expr* )

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

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

42 Constraint: The = *initialization-expr* must appear if the statement contains a PARAMETER attri-  
43 bute (5.1.2.1).

44 Constraint: If = *initialization-expr* appears, a double colon separator must appear before the  
45 *entity-decl-list*.

46 Constraint: The = *initialization-expr* must not appear if *object-name* is a dummy argument, a func-  
47 tion result, an object in a named common block unless the type declaration is in a  
48 block data program unit, an object in blank common, an allocatable array, a pointer,

## Data Object Declarations and Specifications

- 1 an external name, an intrinsic name, or an automatic object.
- 2 Constraint: The \* *char-length* option is permitted only if the type specified is character.
- 3 Constraint: The ALLOCATABLE attribute may be used only when declaring an array that is not  
4 a dummy argument or a function result.
- 5 Constraint: An array declared with a POINTER or an ALLOCATABLE attribute must be speci-  
6 fied with an *array-spec* that is a *deferred-shape-spec-list* (5.1.2.4.3).
- 7 Constraint: An *array-spec* for a *function-name* that does not have the pointer attribute must be an  
8 *explicit-shape-spec-list*.
- 9 Constraint: An *array-spec* for a *function-name* that does have the pointer attribute must be a  
10 *deferred-shape-spec-list*.
- 11 Constraint: If the POINTER attribute is specified, the INTENT, EXTERNAL, or INTRINSIC attri-  
12 bute must not be specified. *TARGET,*
- 13 Constraint: If the TARGET attribute is specified, the EXTERNAL, INTRINSIC, or PARAMETER  
14 attribute must not be specified. *POINTER*
- 15 Constraint: The PARAMETER attribute must not be specified for dummy arguments, pointers,  
16 functions, or objects in a common block.
- 17 Constraint: The INTENT and OPTIONAL attributes may be specified only for dummy argu-  
18 ments.
- 19 Constraint: An entity must not have the PUBLIC attribute if its type has the PRIVATE attribute.
- 20 Constraint: The SAVE attribute must not be specified for an object that is in a common block, a  
21 dummy argument, a procedure, a function result, or an automatic data object.
- 22 Constraint: An entity must not have the EXTERNAL attribute if it has the INTRINSIC attribute.
- 23 Constraint: An entity in a *type-declaration-stmt* must not have the EXTERNAL or INTRINSIC  
24 attribute specified unless it is a function.
- 25 Constraint: An array must not have both the ALLOCATABLE attribute and the POINTER attri-  
26 bute.
- 27 Constraint: An entity must not be given explicitly any attribute more than once in a scoping unit.
- 28 Constraint: The value specified in a *kind-selector* must be nonnegative.
- 29 Note that the double colon separator in a *type-declaration-stmt* is required only if an *attr-spec* or  
30 = *initialization-expr* is specified; otherwise, the separator is optional.
- 31 A name that identifies a specific intrinsic function in a scoping unit has a type as specified in 13.12.  
32 An explicit type declaration statement is not required; however, it is permitted. Specifying a type  
33 for a generic intrinsic function name in a type declaration statement is not sufficient, by itself, to  
34 remove the generic properties from that function.
- 35 The *specification-expr* (7.1.6.2) of a *type-param-value* (5.1.1.5) or an *array-spec* (5.1.2.4) may be a non-  
36 constant expression provided the specification expression is in an interface body (12.3.2.1) or in the  
37 specification part of a subprogram. If the data object being declared depends on the value of such  
38 a nonconstant expression and is not a dummy argument, such an object is called an automatic  
39 data object. An automatic object must not appear in a SAVE or DATA statement nor be declared  
40 with a SAVE attribute nor be initially defined by an = *initialization-expr*.
- 41 If a *length-selector* is a nonconstant expression, the length is declared at the entry of the procedure  
42 and is not affected by any redefinition or undefinition of the variables in the specification expres-  
43 sion during execution of the procedure.
- 44 If an *entity-decl* contains an = *initialization-expr* and the *object-name* does not have the PARAMETER  
45 attribute, *object-name* is a variable whose value is initially defined. The *object-name* becomes  
46 defined with the value determined from *initialization-expr* in accordance with the rules of intrinsic  
47 assignment (7.5.1.4). A variable, or part of a variable, must not be initialized more than once in an

1 executable program.

2 The presence of = *initialization-expr* implies that *object-name* is saved, except for an *object-name* in a  
3 named common block. The implied SAVE attribute may be reaffirmed by explicit use of the SAVE  
4 attribute in the type declaration statement, or by inclusion of the object name in a SAVE statement  
5 (5.2.4).

6 An assumed type parameter is a type parameter of a dummy argument that is specified with an  
7 asterisk *type-param-value* (R509).

8 Examples of type declaration statements are:

9 REAL A (10)  
10 LOGICAL, DIMENSION (5, 5) :: MASK1, MASK2  
11 COMPLEX :: CUBE\_ROOT = (-0.5, 0.866)  
12 INTEGER, PARAMETER :: SHORT = SELECTED\_INT\_KIND (4)  
13 REAL (KIND (0.0D0)) A  
14 REAL (KIND = 2) B  
15 COMPLEX (KIND = KIND (0.0D0)) :: C  
16 INTEGER (SHORT) K ! Range at least -9999 to 9999.  
17 TYPE (PERSON) :: CHAIRMAN

18 **5.1.1 Type Specifiers.** A type specifier specifies the type of all entities declared in an entity dec-  
19 laration list. This type may override or confirm the implicit type indicated by the first letter of the  
20 entity name as declared by the implicit typing rules in effect (5.3).

21 **5.1.1.1 INTEGER.** The INTEGER type specifier specifies that all entities whose names are  
22 declared in this statement are of intrinsic type integer (4.3.1.1). The kind selector, if present, speci-  
23 fies the integer representation method. If the kind selector is absent, the kind type parameter is  
24 KIND (0) and the entities declared are of type default integer. An entity declared with a type spec-  
25 ifier INTEGER (KIND (0)) is of the same kind as one declared with the type specifier INTEGER.

26 **5.1.1.2 REAL.** The REAL type specifier specifies that all entities whose names are declared in this  
27 statement are of intrinsic type real (4.3.1.2). The kind selector, if present, specifies the real approxi-  
28 mation method. If the kind selector is absent, the kind type parameter is KIND (0.0) and the enti-  
29 ties declared are of type default real. An entity declared with a type specifier REAL (KIND (0.0)) is  
30 of the same kind as one declared with the type specifier REAL.

31 **5.1.1.3 DOUBLE PRECISION.** The DOUBLE PRECISION type specifier specifies that entities  
32 whose names are declared in this statement are of intrinsic type double precision real (4.3.1.2). The  
33 kind parameter value is KIND (0.0D0). An entity declared with a type specifier  
34 REAL (KIND (0.0D0)) is of the same kind as one declared with the type specifier DOUBLE PRECI-  
35 SION.

36 **5.1.1.4 COMPLEX.** The COMPLEX type specifier specifies that all entities whose names are  
37 declared in this statement are of intrinsic type complex (4.3.1.3). The kind selector, if present, spec-  
38 ifies the real approximation method of the two real values making up the real and imaginary parts  
39 of the complex value. If the kind selector is absent, the kind type parameter is KIND (0.0) and the  
40 entities declared are of type default complex. An entity declared with a type specifier  
41 COMPLEX (KIND (0.0)) is of the same kind as one declared with the type specifier COMPLEX.

42 **5.1.1.5 CHARACTER.** The CHARACTER type specifier specifies that all entities whose names are  
43 declared in this statement are of intrinsic type character (4.3.2.1).

44 The length selector specifies the length of the character entities. The *\*char-length* may be part of an  
45 *entity-decl*, in which case the length is specified for this single entity and overrides the length speci-  
46 fied in the length selector. If neither a length selector nor a *\*char-length* is specified, the length of  
47 the entity is 1.

48 R506 *char-selector* is *length-selector*

## Data Object Declarations and Specifications

1 or ( [ LEN = ] *type-param-value* , optional only if LEN is absent  
2 [ KIND = ] scalar-int-initialization-expr )  
3 or ( KIND = *scalar-int-initialization-expr* [ , LEN = type-param-value ] )  
4 [ , LEN = type-param-value ] )  
5 R507 *length-selector* is ( [ LEN = ] *type-param-value* )  
6 or \* *char-length* ( ) only on a type decl  
7 R508 *char-length* is ( *type-param-value* )  
8 or *scalar-int-literal-constant*

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

11 R509 *type-param-value* is *specification-expr*  
12 or \*

13 Constraint: A function name must not be declared with an asterisk *char-length* if the function is  
14 an internal or module function, array-valued, pointer-valued, or recursive.

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

- 17 (1) It may be used to declare a dummy argument of a procedure, in which case the dummy  
18 argument assumes the length of the associated actual argument when the procedure is  
19 invoked.
- 20 (2) It may be used to declare a named constant, in which case the length is that of the con-  
21 stant value.
- 22 (3) In an external function, the name of the function itself may be specified with a length  
23 type parameter value of \*; in this case, any scoping unit invoking the function must  
24 declare this function name with a length type parameter value other than \* or access  
25 such a definition by host or use association. When the function is invoked, the length of  
26 the result variable in the function is assumed from the value of this type parameter.

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

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

32 Examples of character type declaration statements are:

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

35 **5.1.1.6 LOGICAL** The LOGICAL type specifier specifies that all entities whose names are  
36 declared in this statement are of intrinsic type logical (4.3.2.2).

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

39 **5.1.1.7 Derived Type.** A TYPE type specifier specifies that all entities whose names are declared  
40 in this statement are of the derived type specified by the *type-name*. The components of each such  
41 entity also are declared to be of the types specified by the corresponding *component-def* statements  
42 of the *derived-type-def* (4.4.1). When a data entity is specified to be of a derived type, the derived  
43 type must have been defined previously by a *derived-type-def* in the scoping unit or be  
44 accessible there by use or host association.  
45 A scalar entity of derived type is a structure. If a derived type has the SEQUENCE property, a sca-  
46 lar entity of the type is a sequence structure. A scalar entity of numeric sequence type (4.4.1) is a  
47 numeric sequence structure. A scalar entity of character sequence type (4.4.1) is a character  
sequence structure.

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

5 **5.1.2 Attributes.** The additional attributes that may appear in the attribute specification of a type  
 6 declaration statement further specify the nature of the entities being declared or specify restric-  
 7 tions on their use in the program.

8 **5.1.2.1 PARAMETER Attribute.** The **PARAMETER** attribute specifies that entities whose names  
 9 are declared in this statement are named constants. The *object-name* becomes defined with the  
 10 value determined from the *initialization-expr* that appears on the right of the equals, in accordance  
 11 with the rules of intrinsic assignment (7.5.1.4). The appearance of a **PARAMETER** attribute in a  
 12 specification requires that the = *initialization-expr* option appear for all objects in the *entity-decl-list*.

13 Any named constant that appears in the initialization expression must have been defined previ-  
 14 ously in the same type declaration statement, defined in a prior **PARAMETER** statement or type  
 15 declaration statement using the **PARAMETER** attribute, or made accessible by use association or  
 16 host association. A named constant must not be referenced in any other context unless it has been  
 17 defined in a prior **PARAMETER** statement or type declaration statement using the **PARAMETER**  
 18 attribute, or made accessible by use association or host association.

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

20 Examples of declarations with a **PARAMETER** attribute are:

```
21 REAL, PARAMETER :: ONE = 1.0, Y = 4.1 / 3.0
22 INTEGER, DIMENSION (3), PARAMETER :: ORDER = (/ 1, 2, 3 /)
```

23 **5.1.2.2 Accessibility Attribute.** The **accessibility** attribute specifies the accessibility of the enti-  
 24 ties in the *entity-decl-list* to other program units by a **USE** statement. This includes derived-type  
 25 definitions in the module.

```
26 R510 access-spec is PUBLIC
27 or PRIVATE
```

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

29 Entities that are declared with a **PRIVATE** attribute are not accessible outside the module. Entities  
 30 that are declared with a **PUBLIC** attribute may be made accessible in other program units by the  
 31 **USE** statement. Entities without an explicitly specified *access-spec* have default accessibility, which  
 32 is **PUBLIC** unless the default has been changed by a **PRIVATE** statement (5.2.3).

33 An example of an accessibility specification is:

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

35 **5.1.2.3 INTENT Attribute.** An **INTENT** attribute specifies the intended use of the dummy argu-  
 36 ment.

```
37 R511 intent-spec is IN
38 or OUT
39 or INOUT
```

~~40 **Constraint:** The **INTENT** attribute may appear only in the *specification-part* of a subprogram or an~~  
~~41 *interface body* (12.3.2.1).~~

42 **Constraint:** The **INTENT** attribute must not be specified for a dummy argument that is a dummy  
 43 procedure or a dummy pointer.

44 The **INTENT** (**IN**) attribute specifies that the dummy argument must not be redefined or become  
 45 undefined during the execution of the procedure.

## Data Object Declarations and Specifications

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

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

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

10 An example of an INTENT specification is:

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

15 **5.1.2.4 DIMENSION Attribute.** The DIMENSION attribute specifies that entities whose names  
16 are declared in this statement are arrays. The rank or the rank and shape are specified by the  
17 *array-spec*, if there is one, in the *entity-decl*, or by the *array-spec* in the DIMENSION attribute other-  
18 wise. An *array-spec* in an *entity-decl* specifies either the rank or the rank and shape for a single  
19 array and overrides the *array-spec* in the DIMENSION attribute. If the DIMENSION attribute is  
20 omitted, an *array-spec* must be specified in the *entity-decl* to declare an array in this statement.

```
21 R512 array-spec                is explicit-shape-spec-list
22                                     or assumed-shape-spec-list
23                                     or deferred-shape-spec-list
24                                     or assumed-size-spec
```

25 Constraint: The maximum rank is seven.

26 Examples of DIMENSION attribute specifications are:

```
27 SUBROUTINE EX (N, A, B, S)
28     REAL, DIMENSION (N, 10) :: W           ! Automatic explicit-shape array
29     REAL A (:), B (0:)                   ! Assumed-shape arrays
30     REAL, POINTER :: D (:, :)           ! Array pointer
31     REAL, DIMENSION (:), POINTER :: P    ! Array pointer
32     REAL, ALLOCATABLE, DIMENSION (:) :: E ! Allocatable array
33     REAL :: S (N, *)                   ! Assumed-size array
```

34 **5.1.2.4.1 Explicit-Shape Array.** An explicit-shape array is a named array that is declared with  
35 an *explicit-shape-spec-list*. This specifies explicit values for the bounds in each dimension of the  
36 array.

```
37 R513 explicit-shape-spec        is [ lower-bound : ] upper-bound
38 R514 lower-bound                is specification-expr
39 R515 upper-bound                is specification-expr
```

40 Constraint: An explicit-shape array whose bounds depend on the values of nonconstant expres-  
41 sions must be a dummy argument, a function result, or an automatic array of a pro-  
42 cedure.

43 An automatic array is an explicit-shape array that is declared in a procedure subprogram, is not a  
44 dummy argument, and has bounds that are nonconstant specification expressions.

45 If an explicit-shape array has bounds that are nonconstant specification expressions, the bounds,  
46 and hence shape, are determined at entry to the procedure by evaluating the bounds expressions.  
47 The bounds of such an array are unaffected by any redefinition or undefinition of the specification  
48 expression variables during execution of the procedure.

1 The values of each *lower-bound* and *upper-bound* determine the bounds of the array along a particu-  
 2 lar dimension and hence the extent of the array in that dimension. The value of a lower bound or  
 3 an upper bound may be positive, negative, or zero. The subscript range of the array in that dimen-  
 4 sion is the set of integer values between and including the lower and upper bounds, provided the  
 5 upper bound is not less than the lower bound. If the upper bound is less than the lower bound,  
 6 the range is empty, the extent in that dimension is zero, and the array is of zero size. If the *lower-*  
 7 *bound* is omitted, the default value is 1. The number of sets of bounds specified is the rank.

8 **5.1.2.4.2 Assumed-Shape Array.** An assumed-shape array is a nonpointer dummy argument  
 9 array that takes its shape from the associated actual argument array.

10 R516 *assumed-shape-spec* is [ *lower-bound* ] :

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

12 The extent of a dimension of an assumed-shape array is the extent of the corresponding dimension  
 13 of the associated actual argument array. If the lower bound value is  $d$  and the extent of the corre-  
 14 sponding dimension of the associated actual argument array is  $s$ , then the value of the upper  
 15 bound is  $s + d - 1$ . The lower bound is *lower-bound*, if present, and 1 otherwise.

16 **5.1.2.4.3 Deferred-Shape Array.** A deferred-shape array is an array pointer or an allocatable  
 17 array.

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

21 The ALLOCATABLE attribute may be specified for an array in a type declaration statement or in  
 22 an ALLOCATABLE statement (5.2.6). An array with the ALLOCATABLE attribute must be  
 23 declared with a *deferred-shape-spec-list* in a type declaration statement, an ALLOCATABLE state-  
 24 ment, or a DIMENSION statement (5.2.5). The type and type parameters may be specified in a  
 25 type declaration statement.

26 An array pointer is an array with the POINTER attribute and a specified rank. Its type, type  
 27 parameters, and rank are specified in a type declaration statement or a component definition state-  
 28 ment, but its bounds, and hence shape, are determined when it is associated with a target by  
 29 pointer assignment (7.5.2) or by execution of an ALLOCATE statement (6.3.1). The POINTER attri-  
 30 bute may be specified for an array in a type declaration statement or in a POINTER statement  
 31 (5.2.7). An array with the POINTER attribute must be declared with a *deferred-shape-spec* in a type  
 32 declaration statement, a POINTER statement, a DIMENSION statement (5.2.5), or a COMMON  
 33 statement (5.5.2). The type and type parameters may be specified in a type declaration statement.

34 R517 *deferred-shape-spec* is :

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

36 The size, bounds, and shape of an unallocated allocatable array are undefined. No part of such an  
 37 array may be defined, nor may any part of it be referenced except as an argument to an intrinsic  
 38 inquiry function that is inquiring about the allocation status or a property of the type or type  
 39 parameters. The lower and upper bounds of each dimension are those specified in the ALLO-  
 40 CATE statement when the array is allocated.

41 The size, bounds, and shape of the target of a disassociated array pointer are undefined. No part  
 42 of such an array may be defined, nor may any part of it be referenced except as an argument to an  
 43 intrinsic inquiry function that is inquiring about argument presence, a property of the type or type  
 44 parameters, or association status. The bounds of each dimension of an array pointer may be speci-  
 45 fied in two ways:

- 46 (1) They are specified in an ALLOCATE statement (6.3.1) when the target is allocated, or
- 47 (2) They are specified in a pointer assignment statement. The lower bound of each dimen-  
 48 sion is the result of the LBOUND function (13.13.52) applied to the corresponding  
 49 dimension of the target. The upper bound of each dimension is the result of the





1 **5.1.2.6 OPTIONAL Attribute.** The **OPTIONAL** attribute may be specified only in the scoping unit  
 2 of a subprogram or an interface block, and may be specified only for dummy arguments. The  
 3 **OPTIONAL** attribute specifies that the dummy argument need not be associated with an actual  
 4 argument in a reference to the procedure (12.5.2.8). The **PRESENT** intrinsic function (13.13.80)  
 5 may be used to determine whether an actual argument has been associated with a dummy argu-  
 6 ment having the **OPTIONAL** attribute.

7 **5.1.2.7 POINTER Attribute.** The **POINTER** attribute specifies that the object must not be refer-  
 8 enced or defined unless, as a result of executing a pointer assignment (7.5.2) or an **ALLOCATE**  
 9 statement (6.3.1), it becomes pointer associated with a target object that may be referenced or  
 10 defined. If the pointer is an array, it must be declared with a *deferred-shape-spec-list*. An object with  
 11 the **POINTER** attribute occupies an unspecified storage unit (14.6.3.1). Examples of **POINTER**  
 12 attribute specifications are:

```
13 TYPE (NODE), POINTER :: CURRENT, TAIL
14 REAL, DIMENSION (:, :), POINTER :: IN, OUT, SWAP
```

15 **5.1.2.8 TARGET Attribute.** The **TARGET** attribute specifies that the object may have a pointer  
 16 associated with it (7.5.2). An object without the **TARGET** or **POINTER** attribute must not have a  
 17 pointer associated with it. Examples of **TARGET** attribute specifications are:

```
18 TYPE (NODE), TARGET :: HEAD
19 REAL, DIMENSION (1000, 1000), TARGET :: A, B
```

20 **5.1.2.9 ALLOCATABLE Attribute.** The **ALLOCATABLE** attribute specifies that objects declared in  
 21 the statement are allocatable arrays. Such arrays must be deferred-shape arrays whose shape is  
 22 determined when space is allocated for each array by the execution of an **ALLOCATE** statement  
 23 (6.3.1).

24 **5.1.2.10 EXTERNAL Attribute.** The **EXTERNAL** attribute specifies that an object name in a decla-  
 25 ration containing this attribute is an external function or a dummy function and permits the name  
 26 to be used as an actual argument. This attribute also may be declared via the **EXTERNAL** state-  
 27 ment (12.3.2.2).

28 **5.1.2.11 INTRINSIC Attribute.** The **INTRINSIC** attribute specifies that an object name in a decla-  
 29 ration containing this attribute must be the specific or generic name of an intrinsic function and  
 30 permits the name to be used as an actual argument if it is a specific name of an intrinsic function  
 31 (13.12). This attribute also may be declared via the **INTRINSIC** statement (12.3.2.3).

32 **5.2 Attribute Specification Statements.** All attributes (other than type) may be specified for  
 33 entities, independently of type, by single attribute specification statements. The combination of  
 34 attributes that may be specified for a particular entity is subject to the same restrictions regardless  
 35 of the method of specification.

### 36 5.2.1 INTENT Statement.

```
37 R519 intent-stmt is INTENT (intent-spec) [ :: ] dummy-arg-name-list
```

38 Constraint: An *intent-stmt* may appear only in the *specification-part* of a subprogram or an inter-  
 39 face body (12.3.2.1).

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

41 This statement specifies the intended use of the specified dummy arguments (5.1.2.3). Each speci-  
 42 fied dummy argument has the **INTENT** attribute.

43 An example of an **INTENT** statement is:

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

## Data Object Declarations and Specifications

### 1 5.2.2 OPTIONAL Statement.

2 R520 *optional-stmt* is OPTIONAL [ :: ] *dummy-arg-name-list*

3 Constraint: An *optional-stmt* may occur only in the scoping unit of a subprogram or an interface  
4 body.

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

8 An example of an OPTIONAL statement is:

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

### 11 5.2.3 Accessibility Statements.

12 R521 *access-stmt* is *access-spec* [ [ :: ] *access-id-list* ]

13 R522 *access-id* is *use-name*  
14 or *generic-spec*

15 Constraint: An *access-stmt* may appear only in the scoping unit of a module. Only one accessibil-  
16 ity statement with an omitted *access-id-list* is permitted in the scoping unit of a mod-  
17 ule.

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

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

23 This statement declares the accessibility, PUBLIC or PRIVATE, of the entities (5.1.2.2). A proce-  
24 dure that has a generic identifier is accessible through the generic identifier even if its specific  
25 name is private. *(12.3.2.1) that is PUBLIC*

26 If an ACCESS statement without an *access-id-list* appears in the scoping unit of a module, the state-  
27 ment sets the default accessibility that applies to all potentially accessible entities in the scoping  
28 unit of the module. The statement

29 PUBLIC

30 sets the default to public accessibility. The statement

31 PRIVATE

32 sets the default to private accessibility. If no such statement appears in a module, the default is  
33 public accessibility.

34 Examples of accessibility statements are:

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

### 38 5.2.4 SAVE Statement.

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

40 R524 *saved-entity* is *object-name*  
41 or / *common-block-name* /

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

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

4 All objects named explicitly or included within a common block named explicitly have the SAVE  
 5 attribute (5.1.2.5). If a particular common block name is specified in a SAVE statement in any scop-  
 6 ing unit of an executable program other than the main program, it must be specified in a SAVE  
 7 statement in every scoping unit in which that common block appears except in the scoping unit of  
 8 the main program. For a common block declared in a SAVE statement, the current values of the  
 9 objects in a common block storage sequence (5.5.2.1) at the time a RETURN or END statement is  
 10 executed are made available to the next scoping unit in the execution sequence of the executable  
 11 program that specifies the common block name or accesses the common block. If a named com-  
 12 mon block is specified in the scoping unit of the main program, the current values of the common  
 13 block storage sequence are made available to each scoping unit that specifies the named common  
 14 block. The definition status of each object in the named common block storage sequence depends  
 15 on the association that has been established for the common block storage sequence.

prohibited  
or not  
required?

16 A SAVE statement with an empty saved entity list is treated as though it contained the names of  
 17 all allowed items in the same scoping unit.

18 A SAVE statement may appear in the specification part of a main program and has no effect.

19 An example of a SAVE statement is:

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

## 21 5.2.5 DIMENSION Statement.

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

24 This statement specifies a list of object names to have the DIMENSION attribute (5.1.2.4) and speci-  
 25 fies the array properties that apply for each object named.

26 An example of a DIMENSION statement is:

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

## 28 5.2.6 ALLOCATABLE Statement.

29 R526 *allocatable-stmt* is ALLOCATABLE [ :: ] *array-name* ■  
 30 ■ [ ( *deferred-shape-spec-list* ) ] ■  
 31 ■ [ , *array-name* [ ( *deferred-shape-spec-list* ) ] ] ...

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

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

35 This statement specifies a list of array names that have the ALLOCATABLE attribute (5.1.2.9). The  
 36 shape of an allocatable array is determined when space is allocated for the array by the execution  
 37 of an ALLOCATE statement (6.3.1).

38 An example of an ALLOCATABLE statement is:

39 REAL A, B (:)  
 40 ALLOCATABLE :: A (:, :), B

## 41 5.2.7 POINTER Statement.

42 R527 *pointer-stmt* is POINTER [ :: ] *object-name* ■  
 43 ■ [ ( *deferred-shape-spec-list* ) ] ■  
 44 ■ [ , *object-name* [ ( *deferred-shape-spec-list* ) ] ] ...

## Data Object Declarations and Specifications

- 1 Constraint: The INTENT attribute must not be specified for an *object-name*.
- 2 Constraint: If the DIMENSION attribute for an *object-name* is specified elsewhere in the scoping  
3 unit, the *array-spec* must be a *deferred-shape-spec-list*.
- 4 Constraint: The PARAMETER attribute must not be specified for *object-name*.
- 5 This statement specifies a list of object names that have the POINTER attribute (5.1.2.7). An object  
6 that has the POINTER attribute must not be referenced or defined unless, as a result of executing a  
7 pointer assignment (7.5.2) or an ALLOCATE statement (6.3.1), it becomes pointer associated with a  
8 target object that may be referenced or defined.
- 9 An example of a POINTER statement is:
- 10 TYPE (NODE) :: CURRENT  
11 POINTER :: CURRENT, A (:, :)

### 12 5.2.8 TARGET Statement.

13 R528 *target-stmt* is TARGET [ :: ] *object-name* [ ( *array-spec* ) ] ■  
14 ■ [ , *object-name* [ ( *array-spec* ) ] ] ...

- 15 Constraint: The PARAMETER attribute must not be specified for an *object-name*.
- 16 This statement specifies a list of object names that have the TARGET attribute and thus may have  
17 pointers associated with them.
- 18 An example of a TARGET statement is:
- 19 TARGET :: A (1000, 1000), B

### 20 5.2.9 DATA Statement. A DATA statement is used to provide initial values for variables.

21 R529 *data-stmt* is DATA *data-stmt-set* [ [ , ] *data-stmt-set* ] ...

- 22 A variable, or part of a variable, must not be initialized more than once in an executable program.
- 23 A variable that appears in a DATA statement and is typed implicitly may appear in a subsequent  
24 type declaration only if that declaration confirms the implicit typing. An array name, array sec-  
25 tion, or array element that appears in a DATA statement must have had its array properties estab-  
26 lished by a previous declaration statement.
- 27 Except for variables in named common blocks, a named variable has the SAVE attribute if any part  
28 of it is initialized in a DATA statement, and this may be confirmed by a SAVE statement or a type  
29 declaration statement containing the SAVE attribute.

30 R530 *data-stmt-set* is *data-stmt-object-list* / *data-stmt-value-list* /

31 R531 *data-stmt-object* is *variable*  
32 or *data-implied-do*

33 R532 *data-stmt-value* is [ *data-stmt-repeat* \* ] *data-stmt-constant*

34 R533 *data-stmt-constant* is *scalar-constant*  
35 or *signed-int-literal-constant*  
36 or *signed-real-literal-constant*  
37 or *structure-constructor*  
38 or *boz-literal-constant*

39 R534 *data-stmt-repeat* is *scalar-int-constant*

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

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

- 1 Constraint: ~~array-element must not be a subobject with a constant parent.~~ <sup>or structure components must not have a constant parent.</sup>
- 2 R537 *data-i-do-variable* is *scalar-int-variable*
- 3 Constraint: *data-i-do-variable* must be a named variable.
- 4 Constraint: The DATA statement repeat factor must be positive or zero. If the DATA statement  
5 repeat factor is a named constant, it must have been declared previously in the scop-  
6 ing unit or made accessible by use association or host association.
- 7 Constraint: If a *data-stmt-constant* is a *structure-constructor*, each component must be ~~a constant~~ <sup>an initialization</sup>  
8 expression.
- 9 Constraint: A variable whose name or designator is included in a *data-stmt-object-list* or a *data-i-*  
10 *do-object-list* must not be: a dummy argument, made accessible by use association or  
11 host association, in a named common block unless the DATA statement is in a block  
12 data program unit, in a blank common block, a function name, a function result  
13 name, an automatic object, a pointer, or an allocatable array.
- 14 Constraint: A subscript in an array element *data-i-do-object* must be an expression whose prima-  
15 ries are either constants or DO variables of the containing *data-implied-dos*.
- 16 Constraint: A *scalar-int-expr* of a *data-implied-do* must involve as primaries only constants or DO  
17 variables of the containing *data-implied-dos*, <sup>and each operation must be intrinsic</sup>
- 18 The *data-stmt-object-list* is expanded to form a sequence of scalar variables. An array whose  
19 unqualified name appears in a *data-stmt-object-list* is equivalent to a complete sequence of its array  
20 elements in array element order (6.2.2.2). An array section is equivalent to the sequence of its  
21 array elements in array element order. A *data-implied-do* is expanded to form a sequence of array  
22 elements, under the control of the implied-DO variable, as in the DO construct (8.1.4.4).
- 23 Note that <sup>and structure components</sup> zero-sized arrays and implied-DO lists with iteration counts of zero contribute no vari-  
24 ables to the expanded sequence of scalar variables, but that a zero-length character variable does  
25 contribute a variable to the list.
- 26 The *data-stmt-value-list* is expanded to form a sequence of scalar constant values. Each such value  
27 must be a constant that is either previously defined or made accessible by a use association or host  
28 association. A data statement repeat factor indicates the number of times the following constant is  
29 to be included in the sequence; omission of a data statement repeat factor has the effect of a repeat  
30 factor of 1. Note that values with a repeat factor of zero contribute no values to the expanded  
31 sequence of scalar constant values.
- 32 The expanded sequences of scalar variables and constant values are in one-to-one correspondence.  
33 Each constant specifies the initial value for the corresponding variable. The lengths of the two  
34 expanded sequences must be the same.
- 35 If an object is of type character or logical, the corresponding constant must be of the same type.  
36 When the object is of type real or complex, the corresponding constant must be of type integer,  
37 real, or complex. When the object is of type integer, the corresponding constant either must be of  
38 type integer, real, or complex, or must be a binary, octal, or hexadecimal literal constant. If an  
39 object is of derived type, the corresponding constant must be of the same type.
- 40 The value of the constant must be compatible with its corresponding variable according to the  
41 rules of intrinsic assignment (7.5.1.4), and the variable becomes initially defined with the value of  
42 the constant in accordance with the rules of intrinsic assignment.
- 43 Within an interface block (12.3.2.1), data initialization by a type declaration statement has no effect.
- 44 Examples of DATA statements are:
- 45 CHARACTER (LEN = 10) NAME  
46 INTEGER, DIMENSION (0:9) :: MILES  
47 REAL, DIMENSION (100, 100) :: SKEW  
48 TYPE (PERSON) MYNAME, YOURNAME

## Data Object Declarations and Specifications

```
1 DATA NAME / 'JOHN DOE' /, MILES / 10 * 0 /
2 DATA ((SKEW (K, J), J = 1, K), K = 1, 100) / 5050 * 0.0 /
3 DATA ((SKEW (K, J), J = K + 1, 100), K = 1, 99) / 4950 * 1.0 /
4 DATA MYNAME / PERSON (21, 'JOHN SMITH') /
5 DATA YOURNAME % AGE, YOURNAME % NAME / 35, 'FRED BROWN' /
```

6 The character variable NAME is initialized with the value JOHN DOE with padding on the right  
7 because the length of the constant is less than the length of the variable. All ten elements of the  
8 integer array MILES are initialized to zero. The two-dimensional array SKEW is initialized so that  
9 the lower triangle of SKEW is zero and the strict upper triangle is one. The structures MYNAME  
10 and YOURNAME are declared using the derived type PERSON from 4.4.1. MYNAME is initial-  
11 ized by a structure constructor. YOURNAME is initialized by supplying a separate value for each  
12 component.

13 **5.2.10 PARAMETER Statement.** The PARAMETER statement provides a means of defining a  
14 named constant. Named constants defined by a PARAMETER statement have exactly the same  
15 properties and restrictions as those declared in a type statement specifying a PARAMETER attri-  
16 bute (5.1.2.1).

17 R538 *parameter-stmt* is PARAMETER (*named-constant-def-list*)

18 R539 *named-constant-def* is *named-constant = initialization-expr*

19 The named constant must have its type, shape, and any type parameters specified either by a pre-  
20 vious occurrence in a type declaration statement in the same scoping unit, or by the implicit typing  
21 rules currently in effect for the scoping unit. If the named constant is typed by the implicit typing  
22 rules, its appearance in any subsequent type declaration statement must confirm this implied type  
23 and the values of any implied type parameters.

24 Each named constant becomes defined with the value determined from the initialization expres-  
25 sion that appears on the right of the equals, in accordance with the rules of intrinsic assignment  
26 (7.5.1.4).

27 A named constant that appears in the initialization expression must have been defined previously  
28 in the same PARAMETER statement, defined in a prior PARAMETER statement or type declara-  
29 tion statement using the PARAMETER attribute, or made accessible by use association or host  
30 association.

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

32 Each named constant has the PARAMETER attribute.

33 An example of a PARAMETER statement is:

34 PARAMETER (MODULUS = MOD (28, 3), NUMBER\_OF\_SENATORS = 100)

35 **5.3 IMPLICIT Statement.** In a scoping unit, an IMPLICIT statement specifies a type, and pos-  
36 sibly type parameters, for all implicitly typed data entities whose names begin with one of the let-  
37 ters specified in the statement. Alternatively, it may indicate that no implicit typing rules are to  
38 apply in a particular scoping unit.

39 R540 *implicit-stmt* is IMPLICIT *implicit-spec-list*  
40 or IMPLICIT NONE

41 R541 *implicit-spec* is *type-spec (letter-spec-list)*

42 R542 *letter-spec* is *letter [- letter]*

43 Constraint: If IMPLICIT NONE is specified in a scoping unit, it must precede any PARAMETER  
44 statements that appear in the scoping unit and there must be no other IMPLICIT  
45 statements in the scoping unit.

46 Constraint: If the minus and second letter appear, the second letter must follow the first letter  
47 alphabetically.

1 A *letter-spec* consisting of two letters separated by a minus is equivalent to writing a list containing  
 2 all of the letters in alphabetical order in the alphabetic sequence from the first letter through the  
 3 second letter. For example, A-C is equivalent to A, B, C. The same letter must not appear as a single  
 4 letter, or be included in a range of letters, more than once in all of the IMPLICIT statements in a  
 5 scoping unit.

6 In each scoping unit, there is a mapping, which may be null, between each of the letters A, B, ..., Z  
 7 and a type (and type parameters). An IMPLICIT statement specifies the mapping for the letters in  
 8 its *letter-spec-list*. IMPLICIT NONE specifies the null mapping for all the letters. If a mapping is  
 9 not specified for a letter, the default is the mapping in the host scoping unit. A program unit is  
 10 treated as if it had a host with the declaration

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

12 Any data entity that is not explicitly declared by a type declaration statement, is not an intrinsic  
 13 function, and is not made accessible by use association or host association is declared implicitly to  
 14 be of the type (and type parameters) mapped from the first letter of its name, provided the map-  
 15 ping is not null. The data entity is treated as if it were declared in an explicit type declaration in  
 16 the outermost scoping unit in which it appears. An explicit type specification in a FUNCTION  
 17 statement overrides an IMPLICIT statement for the name of that function subprogram.

18 The following are examples of the use of IMPLICIT statements:

```
19 MODULE EXAMPLE_MODULE
20   IMPLICIT NONE
21   ...
22   INTERFACE
23     FUNCTION FUN (I)      ! All data entities must
24       INTEGER FUN, I     ! be declared explicitly
25   END FUNCTION FUN
26 END INTERFACE
27 CONTAINS
28   FUNCTION JFUN (J)      ! All data entities must
29     INTEGER JFUN, J     ! be declared explicitly.
30     ...
31   END FUNCTION JFUN
32 END MODULE EXAMPLE_MODULE

33 SUBROUTINE SUB
34   IMPLICIT COMPLEX (C)
35   C = (3.0, 2.0)        ! C is implicitly declared COMPLEX
36   ...
37 CONTAINS
38   SUBROUTINE SUB1
39     IMPLICIT INTEGER (A, C)
40     C = (0.0, 0.0)      ! C is host associated and of
41                       ! type complex
42     Z = 1.0             ! Z is implicitly declared REAL
43     A = 2               ! A is implicitly declared INTEGER
44     CC = 1              ! CC is implicitly declared INTEGER
45     ...
46   END SUBROUTINE SUB1

47   SUBROUTINE SUB2
48     Z = 2.0             ! Z is implicitly declared REAL and
49                       ! is different from the variable of
50                       ! the same name in SUB1
51     ...
52   END SUBROUTINE SUB2
```

## Data Object Declarations and Specifications

```
1      SUBROUTINE SUB3
2          USE EXAMPLE_MODULE ! Accesses integer function FUN
3                                ! by use association
4          Q = FUN (K)          ! Q is implicitly declared REAL and
5          ...                 ! K is implicitly declared INTEGER
6      END SUBROUTINE SUB3
7  END SUBROUTINE SUB
```

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

```
10  IMPLICIT TYPE (POSN) (A-B, W-Z), INTEGER (C-V)
11  TYPE POSN
12      REAL X, Y
13      INTEGER Z
14  END TYPE POSN
```

15 variables beginning with the letters A, B, W, X, Y, and Z are implicitly typed with the type POSN  
16 and the remaining variables are implicitly typed with type INTEGER.

17 **5.4 NAMELIST Statement.** A NAMELIST statement specifies a group of named data objects  
18 which can then be referred to by a single name for the purpose of data transfer (9.4, 10.9).

```
19  R543  namelist-stmt          is NAMELIST / namelist-group-name / ■
20                                ■ namelist-group-object-list ■
21                                ■ [, ] / namelist-group-name / ■
22                                ■ namelist-group-object-list ] ...
```

```
23  R544  namelist-group-object is variable-name
```

24 Constraint: A *namelist-group-object* must not be an array dummy argument with nonconstant  
25 bounds, a variable with assumed character length, an automatic object, a pointer, of a  
26 type that has an ultimate component that is a pointer, an allocatable array, or a sub-  
27 object of any of the preceding objects.

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

30 The order in which the data objects (variables) are specified in the NAMELIST statement deter-  
31 mines the order in which the values appear on output.

32 Any *namelist-group-name* may occur in more than one NAMELIST statement in a scoping unit. The  
33 *namelist-group-object-list* following each successive appearance of the same *namelist-group-name* in a  
34 scoping unit is treated as a continuation of the list for that *namelist-group-name*.

35 A namelist group object may be a member of more than one namelist group.

36 A namelist group object either must have its type, type parameters, and shape specified by a previ-  
37 ous occurrence in a type declaration statement in the same scoping unit, or must be determined by  
38 the implicit typing rules currently in effect for the scoping unit. If a namelist group object is typed  
39 by the implicit typing rules, its appearance in any subsequent type declaration statement must  
40 confirm this implied type.

41 An example of a NAMELIST statement is:

```
42  NAMELIST /NLIST/ A, B, C
```

43 **5.5 Storage Association of Data Objects.** In general, the physical storage units or storage  
44 order for data objects is not specifiable. However, the EQUIVALENCE statement, the COMMON  
45 statement, and the SEQUENCE statement provide for control of the order and layout of storage  
46 units. The general mechanism of storage association is described in 14.6.3.



1 **5.5.1 EQUIVALENCE Statement.** An EQUIVALENCE statement is used to specify the sharing of  
 2 storage units by two or more objects in a scoping unit. This causes storage association of the  
 3 objects that share the storage units.

4 If the equivalenced objects have differing type or type parameters, the EQUIVALENCE statement  
 5 does not cause type conversion or imply mathematical equivalence. If a scalar and an array are  
 6 equivalenced, the scalar does not have array properties and the array does not have the properties  
 7 of a scalar.

8 R545 *equivalence-stmt* is EQUIVALENCE *equivalence-set-list*  
 9 R546 *equivalence-set* is ( *equivalence-object* , *equivalence-object-list* )  
 10 R547 *equivalence-object* is *variable-name*  
 11 or *array-element*  
 12 or *substring*

13 Constraint: An *equivalence-object* must not be a dummy argument, a pointer, an allocatable array,  
 14 an object of a nonsequence derived type or of a sequence derived type containing a  
 15 pointer, an automatic object, a function name, an entry name, a result name, a named  
 16 constant, or a subobject of any of the preceding objects.

17 Constraint: Each subscript or substring range expression in an *equivalence-object* must be an integer  
 18 initialization expression (7.1.6.1).  
*at only level of component selection*

19 Constraint: If an *equivalence-object* is of type default integer, default real, double precision real,  
 20 default complex, default logical, or numeric sequence type, all of the objects in the  
 21 equivalence set must be of these types.

22 Constraint: If an *equivalence-object* is of type default character or character sequence type, all of  
 23 the objects in the equivalence set must be of these types.

24 Constraint: If an *equivalence-object* is of a derived type that is not a numeric sequence or character  
 25 sequence type, all of the objects in the equivalence set must be of the same type.

26 Constraint: If an *equivalence-object* is of an intrinsic type other than default integer, default real,  
 27 double precision real, default complex, default logical, or default character, all of the  
 28 objects in the equivalence set must be of the same type with the same kind type  
 29 parameter value.

30 **5.5.1.1 Equivalence Association.** An EQUIVALENCE statement specifies that the storage  
 31 sequences (14.6.3.1) of the data objects specified in an *equivalence-set* are storage associated. All of  
 32 the nonzero-sized sequences in the *equivalence-set*, if any, have the same first storage unit, and all of  
 33 the zero-sized sequences in the *equivalence-set*, if any, are storage associated with one another and  
 34 with the first storage unit of any nonzero-sized sequences. This causes the storage association of  
 35 the data objects in the *equivalence-set* and may cause storage association of other data objects.

36 **5.5.1.2 Equivalence of Character Objects.** A data object of type <sup>Default</sup>character may be equiva-  
 37 lenced only with other objects of type <sup>Default</sup>character. The lengths of the equivalenced objects are not  
 38 required to be the same. <sub>Default</sub>

39 An EQUIVALENCE statement specifies that the storage sequences of all the character data objects  
 40 specified in an *equivalence-set* are storage associated. All of the nonzero-sized sequences in the  
 41 *equivalence-set*, if any, have the same first character storage unit, and all of the zero-sized sequences  
 42 in the *equivalence-set*, if any, are storage associated with one another and with the first character  
 43 storage unit of any nonzero-sized sequences. This causes the storage association of the data objects  
 44 in the *equivalence-set* and may cause storage association of other data objects. For example, using  
 45 the declarations:

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

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

## Data Object Declarations and Specifications

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

5 **5.5.1.3 Array Names and Array Element Designators.** For a nonzero-sized array, the use of the  
6 array name unqualified by a subscript list in an EQUIVALENCE statement has the same effect as  
7 using an array element designator that identifies the first element of the array.

8 **5.5.1.4 Restrictions on EQUIVALENCE Statements.** An EQUIVALENCE statement must not  
9 specify that the same storage unit is to occur more than once in a storage sequence. For example,

```
10 REAL, DIMENSION (2) :: A
11 REAL :: B
12 EQUIVALENCE (A (1), B), (A (2), B) ! Not standard conforming
```

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

```
16 REAL A (2)
17 DOUBLE PRECISION D (2)
18 EQUIVALENCE (A (1), D (1)), (A (2), D (2)) ! Not standard conforming
```

19 An EQUIVALENCE statement must not specify the sharing of storage units between objects  
20 declared in different scoping units. For example, the following is prohibited:

```
21 SUBROUTINE A
22     USE MODULE, ONLY : XX
23     INTEGER BB, EE
24     EQUIVALENCE (EE, XX) ! Not standard conforming
25     ...
26     CONTAINS
27         SUBROUTINE C
28             REAL DD
29             EQUIVALENCE (BB, DD) ! Not standard conforming
30         END SUBROUTINE C
31 END SUBROUTINE A
```

32 **5.5.2 COMMON Statement.** The COMMON statement specifies blocks of physical storage,  
33 called **common blocks**, that may be accessed by any of the scoping units in an executable program.  
34 Thus, the COMMON statement provides a global data facility based on storage association (14.6.3).  
35 The common blocks specified by the COMMON statement may be named and are called **named**  
36 **common blocks**, or may be unnamed and are called **blank common**.

```
37 R548 common-stmt is COMMON [ / [ common-block-name ] / ] ■
38                       ■ common-block-object-list ■
39                       ■ [ [ , ] / [ common-block-name ] / ■
40                       ■ common-block-object-list ] ...
```

```
41 R549 common-block-object is variable-name [ ( explicit-shape-spec-list ) ]
```

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

44 **Constraint:** A *common-block-object* must not be a dummy argument, an allocatable array, an auto-  
45 matic object, a function name, an entry name, or a result name.

46 **Constraint:** Each bound in the *explicit-shape-spec* must be a constant specification expression  
47 (7.1.6.2).

48 **Constraint:** If a *common-block-object* is of a derived type, it must be a sequence type (4.4.1).

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

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

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

14 The form *variable-name (explicit-shape-spec-list)* declares *variable-name* to have the DIMENSION attri-  
15 bute and specifies the array properties that apply. If derived-type objects of numeric sequence  
16 type (4.4.1) or character sequence type (4.4.1) appear in common, it is as if the individual compo-  
17 nents were enumerated directly in the common list.

18 Examples of COMMON statements are:

```
19 COMMON /BLOCKA/ A, B, D (10, 30)
20 COMMON I, J, K
```

21 **5.5.2.1 Common Block Storage Sequence.** For each common block, a common block storage  
22 sequence is formed as follows:

- 23 (1) A storage sequence is formed consisting of the sequence of storage units contained in  
24 the storage sequences (14.6.3.1) of all data objects in the common block object lists for  
25 the common block. The order of the storage sequences is the same as the order of the  
26 appearance of the common block object lists in the scoping unit.
- 27 (2) The storage sequence formed in (1) is extended to include all storage units of any stor-  
28 age sequence associated with it by equivalence association. The sequence may be  
29 extended only by adding storage units beyond the last storage unit. Data objects associ-  
30 ated with an entity in a common block are considered to be in that common block.

31 **5.5.2.2 Size of a Common Block.** The size of a common block is the size of its common block  
32 storage sequence, including any extensions of the sequence resulting from equivalence association.

33 **5.5.2.3 Common Association.** Within an executable program, the common block storage  
34 sequences of all nonzero-sized common blocks with the same name have the same first storage  
35 unit, and the common block storage sequences of all zero-sized common blocks with the same  
36 name are storage associated with one another. Within an executable program, the common block  
37 storage sequences of all nonzero-sized blank common blocks have the same first storage unit and  
38 the storage sequences of all zero-sized blank common blocks are associated with one another and  
39 with the first storage unit of any nonzero-sized blank common blocks. This results in the associa-  
40 tion of objects in different scoping units.

41 A nonpointer object of default integer type, default real type, double precision real type, default  
42 complex type, default logical type, or numeric sequence type must become associated only with  
43 nonpointer objects of these types.

44 A nonpointer object of type default character or character sequence type must become associated  
45 only with nonpointer objects of these types.

46 A nonpointer object of a derived type that is not a numeric sequence or character sequence type  
47 must become associated only with nonpointer objects of the same type.

48 A nonpointer object of intrinsic type other than default integer, default real, double precision real,  
49 default complex, default logical, or default character must become associated only with nonpointer

## Data Object Declarations and Specifications

- 1 objects of the same type and type parameter.
- 2 A pointer must become associated only with pointers of the same type, type parameters, and rank.
- 3 **5.5.2.4 Differences between Named Common and Blank Common.** A blank common block
- 4 has the same properties as a named common block, except for the following:

- 5 (1) Execution of a RETURN or END statement may cause data objects in a named common
- 6 block to become undefined unless the common block name has been declared in a
- 7 SAVE statement, but never causes data objects in blank common to become undefined
- 8 (14.7.6).
- 9 (2) Named common blocks of the same name must be of the same size in all scoping units
- 10 of an executable program in which they appear, but blank common blocks may be of
- 11 different sizes.
- 12 (3) A data object in a named common block may be initially defined by means of a DATA
- 13 statement or type declaration statement in a block data program unit, but objects in
- 14 blank common must not be initially defined (11.4).

- 15 **5.5.2.5 Restrictions on Common and Equivalence.** An EQUIVALENCE statement must not
- 16 cause the storage sequences of two different common blocks to be associated. Equivalence associa-
- 17 tion must not cause a common block storage sequence to be extended by adding storage units pre-
- 18 ceding the first storage unit of the first object specified in a COMMON statement for the common
- 19 block. For example, the following is not permitted:

```
20 COMMON /X/ A  
21 REAL B (2)  
22 EQUIVALENCE (A, B (2)) ! Not standard conforming
```

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

## 6. USE OF DATA OBJECTS

1 The appearance of a data object name or subobject designator in a context that requires its value is  
2 termed a reference. A reference is permitted only if the data object is defined. A reference to a  
3 pointer is permitted only if the pointer is associated with a target object that is defined. A data  
4 object becomes defined with a value when the data object name or subobject designator appears in  
5 certain contexts and when certain events occur (14.7).

6 R601 *variable* is *scalar-variable-name*  
7 or *array-variable-name*  
8 or *subobject*

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

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

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

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

14 R602 *subobject* is *array-element*  
15 or *array-section*  
16 or *structure-component*  
17 or *substring*

18 R603 *logical-variable* is *variable*

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

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

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

22 R605 *char-variable* is *variable*

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

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

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

26 R607 *int-variable* is *variable*

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

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

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

30 Pointers and allocatable arrays must not be defined in circumstances explained in 5.1.2.4.3.  
31 Dummy arguments or variables associated with dummy arguments must not be defined in cir-  
32 cumstances explained in 12.5.2.1 and 12.5.2.8.

33 A literal constant is a scalar denoted by a syntactic form which indicates its type, type parameters,  
34 and value. A named constant is a constant that has been associated with a name with the PARAM-  
35 ETER attribute (5.1.2.1, 5.2.10). A reference to a constant is always permitted; redefinition of a con-  
36 stant is never permitted.

37 For example, given the declarations:

38 CHARACTER (10) A, B (10)  
39 TYPE (PERSON) P ! See 4.4.1

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

## Use of Data Objects

1 **6.1 Scalars.** A scalar (2.4.6) is a data entity that can be represented by a single value of the data  
2 type and that is not an array (6.2). Its value, if defined, is a single element from the set of values  
3 that characterize its data type.

4 A scalar has rank zero.

5 **6.1.1 Substrings.** A substring is a contiguous portion of a character string (4.3.2.1). The follow-  
6 ing rules define the forms of a substring:

7 R609 *substring* is *parent-string* ( *substring-range* )

8 R610 *parent-string* is *scalar-variable-name*

9 or *array-element*

10 or *scalar-structure-component*

11 or *scalar-constant*

12 R611 *substring-range* is [ *scalar-int-expr* ] : [ *scalar-int-expr* ]

13 Constraint: *parent-string* must be of type character.

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

17 Let the characters in the parent string be numbered 1, 2, 3, ..., *n*, where *n* is the length of the parent  
18 string. Then the characters in the substring are those from the parent string from the starting point  
19 and proceeding in sequence up to and including the ending point. Both the starting point and the  
20 ending point must be within the range 1, 2, ..., *n* unless the starting point exceeds the ending point,  
21 in which case the substring has length zero. If the starting point is not specified, the default value  
22 is 1. If the ending point is not specified, the default value is *n*.

23 If the parent is a variable, the substring is also a variable.

24 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

29 **6.1.2 Structure Components.** A *structure-component* is one of the components of a structure or is  
30 an array whose elements are components of an array of derived type.

31 R612 *data-ref* is *part-ref* % *part-ref* . [ *part-ref* ] ...

32 R613 *part-ref* is *part-name* [ ( *section-subscript-list* ) ]

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

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

36 Constraint: In a *part-ref* containing a *section-subscript-list*, the number of *section-subscripts* must  
37 equal the rank of *part-name*.

38 The rank of a *part-ref* of the form *part-name* is the rank of *part-name*. The rank of a *part-ref* that has a  
39 section subscript list is the number of subscript triplets and vector subscripts in the list.

40 Constraint: In a *data-ref*, there must not be more than one *part-ref* with nonzero rank. A *part-name* %  
the right of a *part-ref* with nonzero rank must not have the *POINTER* attribute.

41 The rank of a *data-ref* is the rank of the *part-ref* with nonzero rank, if any; otherwise, the rank is  
42 zero. The parent object is a *data-ref* is the data object whose name is the leftmost *part name*.

43 R614 *structure-component* is *data-ref* *conflicts w/ R612*

44 Constraint: In a *structure-component*, there must be more than one *part-ref* and the rightmost *part-ref*  
45 must be of the form *part-name*. - !!! why?

1 The type and type parameters, if any, of a structure component are those of the rightmost part  
 2 name. A structure component must not be referenced or defined before the declaration of the par-  
 3 ent object. A structure component has the INTENT, TARGET, or PARAMETER attribute if the  
 4 parent object has the attribute. A structure component is a pointer only if the rightmost part name  
 5 is defined to have the pointer attribute.

6 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

10 **6.2 Arrays.** An array is a set of scalar data, all of the same type and type parameters, whose  
 11 individual elements are arranged in a rectangular pattern. The scalar data that make up an array  
 12 are the array elements.

13 No order of reference to the elements of an array is indicated by the appearance of the array name  
 14 or designator, except where array element ordering (6.2.2.2) is specified.

15 **6.2.1 Whole Arrays.** A whole array is a named array.

16 A whole array is either a named constant or variable. A whole array named constant is the name  
 17 of a constant expression (5.1.2.1 and 5.2.10) that is an array. A whole array variable is the name of  
 18 a variable that is an array; the name does not have a subscript list appended to it.

19 The appearance of a whole array variable in an executable construct specifies all the elements of  
 20 the array (2.4.7). An assumed-size array is permitted to appear as a whole array in an executable  
 21 construct only as an actual argument in a procedure reference that does not require the shape.

22 The appearance of a whole array name in a nonexecutable statement specifies the entire array.

23 **6.2.2 Array Elements and Array Sections.**

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

25 Constraint: In an *array-element*, every *part-ref* must have rank zero and ~~at least one~~ *part-ref* must  
 26 contain a *subscript-list*. conflicts w/  
the last 6-2-44-45

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

28 Constraint: In an *array-section*, <sup>exactly</sup> one *part-ref* must have a *section-subscript-list* <sup>nonzero rank, and either the final part-ref</sup> ~~and have~~ nonzero  
 29 rank, <sup>or another part-ref has nonzero rank.</sup> ~~and have~~ nonzero rank. <sup>has with</sup>

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

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

33 R618 *section-subscript* is *subscript*  
 34 or *subscript-triplet*  
 35 or *vector-subscript*

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

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

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

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

40 Constraint: The second *subscript* must not be omitted from a *subscript-triplet* in the last dimension  
 41 of an assumed-size array.

42 An array element is a scalar. An array section is an array. If a *substring-range* is present in an  
 43 *array-section*, ~~the object is an array of the shape specified by the section-subscript-list and each ele-~~  
 44 ment is the designated substring of the corresponding element of the array section. For example,

## Use of Data Objects

- 1 with the declarations:  
 2 REAL A (10, 10)  
 3 CHARACTER (LEN = 10) B (5, 5, 5)  
 4 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  
 5 shape (5, 5, 5) whose elements are substrings of length 2 of the corresponding elements of B.  
 6 Examples of array elements and array sections are: *An array element or an array section has the INTENT, TARGET or PARAMETER attribute if its parent does, but it never has the POINTER attribute.*

```

ARRAY_A (1:N:2) % ARRAY_B (I, J) % STRING (K) (:): array section
SCALAR_PARENT % ARRAY_FIELD (J) array element component
SCALAR_PARENT % ARRAY_FIELD (1:N) array section component
SCALAR_PARENT % ARRAY_FIELD (1:N) % SCALAR_FIELD component of an
array section parent
  
```

- 12 **6.2.2.1 Array Elements.** The value of a subscript in an array element must be within the bounds  
 13 for that dimension.

- 14 **6.2.2.2 Array Element Order.** The elements of an array form a sequence known as the array ele-  
 15 ment order. The position of an array element in this sequence is determined by the subscript order  
 16 value of the subscript list designating the element. The subscript order value is computed from the  
 17 formulas in Table 6.1.

- 18 Table 6.1 Subscript Order Value

Rank	Subscript Bounds	Subscript List	Subscript Order Value
1	$j_1:k_1$	$s_1$	$1 + (s_1 - j_1)$
2	$j_1:k_1, j_2:k_2$	$s_1, s_2$	$1 + (s_1 - j_1)$ $+ (s_2 - j_2) \times d_1$
3	$j_1:k_1, j_2:k_2, j_3:k_3$	$s_1, s_2, s_3$	$1 + (s_1 - j_1)$ $+ (s_2 - j_2) \times d_1$ $+ (s_3 - j_3) \times d_2 \times d_1$
.	.	.	.
.	.	.	.
.	.	.	.
7	$j_1:k_1, \dots, j_7:k_7$	$s_1, \dots, s_7$	$1 + (s_1 - j_1)$ $+ (s_2 - j_2) \times d_1$ $+ (s_3 - j_3) \times d_2 \times d_1$ $+ \dots$ $+ (s_7 - j_7) \times d_6$ $\times d_5 \times \dots \times d_1$

- 46 Notes for Table 6.1:

- 47 (1)  $d_i = \max(k_i - j_i + 1, 0)$  is the size of the  $i$ th dimension.  
 48 (2) If the size of the array is nonzero,  $j_i \leq s_i \leq k_i$  for all  $i = 1, 2, \dots, 7$ .

- 49 **6.2.2.3 Array Sections.** An array section is an array subobject ~~designated by an array name or~~  
 50 ~~designator with a section subscript list, optionally followed by a substring range.~~  
*In an array section having a section-subscript-list,*

- 51 *each subscript-triplet and vector-subscript* in the section subscript list indicates a sequence of sub-  
 52 *scripts* which may be empty (6.2.2). Each subscript in such a sequence must be within the bounds



1 for its dimension unless the sequence is empty. The array section is the set of elements from the  
 2 array determined by all possible subscript lists obtainable from the single subscripts or sequences  
 3 of subscripts specified by each section subscript.  
 4 *in an array section with no section subscript list, the rank and shape of the array is the rank and*  
~~4 The rank of the array section is the number of subscript triplets and vector subscripts in the section~~  
 5 *shape of the part not within non-zero rank otherwise*  
 5 subscript list. The shape is the rank-one array whose *i*th element is the number of integer values in  
 6 the sequence indicated by the *i*th subscript triplet or vector subscript. If any of these sequences is  
 7 empty, the array section has size zero. The subscript order of the elements of an array section is  
 8 that of the array data object that the array section represents.

9 **6.2.2.3.1 Subscript Triplet.** A subscript triplet designates a regular sequence of subscripts consist-  
 10 ing of zero or more subscript values. The third expression in the subscript triplet is the increment  
 11 between the subscript values and is called the **stride**. The subscripts and stride of a subscript tri-  
 12 plet are optional. An omitted first subscript in a subscript triplet is equivalent to a subscript whose  
 13 value is the lower bound for the array and an omitted second subscript is equivalent to the upper  
 14 bound. An omitted stride is equivalent to a stride of 1.

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

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

20 The stride must not be zero.

21 For example, suppose an array is declared as A (5, 4, 3). The section A (3 : 5, 2, 1 : 2) is the array of  
 22 shape (3, 2):

```
23     A (3, 2, 1)  A (3, 2, 2)
24     A (4, 2, 1)  A (4, 2, 2)
25     A (5, 2, 1)  A (5, 2, 2)
```

26 When the stride is negative, the sequence begins with the first subscript and proceeds in incre-  
 27 ments of the stride down to the smallest such integer equal to or greater than the second subscript;  
 28 the sequence is empty if the second subscript is greater than the first. For example, if an array is  
 29 declared B (10), the section B (9 : 1 : -2) is the array of shape (5) whose elements are B (9), B (7),  
 30 B (5), B (3), and B (1), in that order.

31 Note that a subscript in a subscript triplet need not be within the declared bounds for that dimen-  
 32 sion if all values used in selecting the array elements are within the declared bounds.

33 For example, if an array is declared as B (10), the array section B (3 : 11 : 7) is the array of shape (2)  
 34 consisting of the elements B (3) and B (10), in that order.

35 **6.2.2.3.2 Vector Subscript.** A vector subscript designates a sequence of subscripts correspond-  
 36 ing to the values of the elements of the expression. Each element of the expression must be  
 37 defined. A **many-one array section** is an array section with a vector subscript having two or more  
 38 elements with the same value. A many-one array section must not appear on the left of the equals  
 39 in an assignment statement or as an input item in a READ statement.

40 For example, suppose Z is a two-dimensional array of shape (5, 7) and U and V are one-  
 41 dimensional arrays of shape (3) and (4), respectively. Assume the values of U and V are:

```
42     U = (/ 1, 3, 2 /)
43     V = (/ 2, 1, 1, 3 /)
```

44 Then Z (3, V) consists of the elements from the third row of Z in the order:

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

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

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

## Use of Data Objects

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

2           Z (1, 2)  Z (1, 1)  Z (1, 1)  Z (1, 3)  
3           Z (3, 2)  Z (3, 1)  Z (3, 1)  Z (3, 3)  
4           Z (2, 2)  Z (2, 1)  Z (2, 1)  Z (2, 3)

5 Because Z (3, V) and Z (U, V) contain duplicate elements from Z, the sections Z (3, V) and Z (U, V)  
6 must not be redefined as sections.

7 An internal file must not be an array section with a vector subscript. An array section with a vec-  
8 tor subscript must not be argument associated with a dummy array that is defined or redefined.  
9 An array section with a vector subscript must not be the target in a pointer assignment statement.

10 **6.3 Dynamic Association.** Dynamic control over the creation, association, and deallocation of  
11 pointer targets is provided by the ALLOCATE, NULLIFY, and DEALLOCATE statements and  
12 pointer assignment. ALLOCATE (6.3.1) creates targets for pointers; pointer assignment (7.5.2)  
13 associates pointers with existing targets; NULLIFY (6.3.2) disassociates pointers from targets, and  
14 DEALLOCATE (6.3.3) deallocates targets. Dynamic association applies to scalars and arrays of  
15 any type. *The ALLOCATE and DEALLOCATE statements are also used to create and deallocate arrays with the ALLOCATABLE attribute.*

16 **6.3.1 ALLOCATE Statement.** The ALLOCATE statement dynamically creates pointer targets and  
17 allocatable arrays.

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

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

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

23 R624   *allocation*                           is *allocate-object* [ ( <sup>*allocate*</sup>~~explicit-shape-spec~~ ) ]

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

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

27 Constraint: A bound in an ~~allocation-explicit-shape-spec~~ must not be an expression involving as a  
28 primary an array inquiry function (13.10.15) whose argument is any other object in  
29 the same ALLOCATE statement.

30 Constraint: The number of <sup>*allocate*</sup>~~explicit-shape-specs~~ in an ~~allocation-explicit-shape-spec-list~~ must be the  
31 same as the rank of the pointer or allocatable array.

32 An example of an ALLOCATE statement is:

33 ALLOCATE (X (N), B (-3 : M, 0:9), STAT = IERR\_ALLOC)

34 At the time an ALLOCATE statement is executed for an array, the values of the lower bound and  
35 upper bound expressions in an explicit-shape specification (5.1.2.4.1) determine the bounds of the  
36 array. Subsequent redefinition or undefinition of any entities in the bound expressions do not  
37 affect the array shape. *Note that zero-length character objects and zero-sized arrays are permitted*  
38 *in the ALLOCATE statement.* *changed*

39 If the STAT= specifier is present, successful execution of the ALLOCATE statement causes the  
40 *stat-variable* to become defined with a value of zero. If an error condition occurs during the execu-  
41 tion of the ALLOCATE statement, the *stat-variable* becomes defined with a processor-dependent  
42 positive integer value.

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



## Use of Data Objects

- 1 (1) An allocatable array with the SAVE attribute,
- 2 (2) An allocatable array in the scoping unit of a module if the module also is accessed by
- 3 another scoping unit that is currently in execution, or
- 4 (3) An allocatable array accessible by host association.
- 5 Any other allocatable array that is currently allocated becomes undefined and the allocation status
- 6 becomes undefined at the execution of a RETURN or END statement.
- 7 If an allocatable *array-name* has an undefined allocation status, the allocatable *array-name* must not
- 8 be subsequently referenced, defined, allocated, or deallocated.

9 **6.3.3.2 Deallocation of Pointer Targets.** If a pointer appears in a DEALLOCATE statement, its  
10 association status must be defined. Deallocating a pointer that is disassociated or whose target  
11 was not created by an ALLOCATE statement causes an error condition in the DEALLOCATE  
12 statement. If a pointer is currently associated with an allocatable array, the pointer must not be  
13 deallocated.

14 A pointer that is not currently associated with the whole of an allocated target object must not be  
15 deallocated. If a pointer is currently associated with a portion (2.4.3.2) of a target object that is  
16 independent of any other portion of the target object, it must not be deallocated. Deallocating a  
17 pointer target causes the pointer association status of any other pointer that is associated with the  
18 target to become undefined.

19 *For a portion of a target*  
When the execution of a procedure is terminated by execution of a RETURN or END statement,  
20 the pointer association status of a pointer declared or accessed in the procedure becomes unde-  
21 fined unless it is one of the following:

- belongs on ch. 14?*
- 22 (1) A pointer with the SAVE attribute,
  - 23 (2) A pointer in blank common,
  - 24 (3) A pointer in a named common block that appears in at least one other scoping unit that
  - 25 is currently in execution,
  - 26 (4) A pointer declared in the scoping unit of a module if the module also is accessed by
  - 27 another scoping unit that is currently in execution,
  - 28 (5) A pointer accessed by host association, or
  - 29 (6) A pointer that is the return value of a function declared to have the POINTER attribute.

*and which does not refer associated to an object that becomes undefined. See 12-1:46-1*

*(1) a pointer associated to a pointer dummy argument and... cf 12-9:23 #*

## 7. EXPRESSIONS AND ASSIGNMENT

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

3 **7.1 Expressions.** An expression represents either a data reference or a computation, and its  
4 value is either a scalar or an array. An expression is formed from operands, operators, and paren-  
5 theses. Simple forms of an operand are constants and variables, such as:

6 3.0  
7 .FALSE.  
8 A  
9 B (I)  
10 C (I:J)

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

13 Examples of intrinsic operators are:

14 +  
15 \*  
16 >  
17 .AND.

18 **7.1.1 Form of an Expression.** Evaluation of an expression produces a value, which has a type,  
19 type parameters (if appropriate), and a shape (7.1.4).

20 Examples of expressions are:

21 A + B  
22 (A - B) \* C  
23 A \*\* B  
24 C .AND. D  
25 F // G

26 An expression is defined in terms of several categories: primary, level-1 expression, level-2 expres-  
27 sion, level-3 expression, level-4 expression, and level-5 expression.

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

31 **7.1.1.1 Primary.**

32 R701 *primary* is *constant*  
33 or *constant-subobject*  
34 or *variable*  
35 or *array-constructor*  
36 or *structure-constructor*  
37 or *function-reference*  
38 or (*expr*)  
39 R702 *constant-subobject* is *subobject*

40 Constraint: *subobject* must be a subobject designator whose parent is a constant.

41 Constraint: A *variable* that is a *primary* must not be an assumed-size array.

42 Examples of a *primary* are:

Example	Syntactic Class
1.0	<i>constant</i>
' ABCDEFGHIJKLMNOPQRSTUVWXYZ' (I:I)	<i>constant-subobject</i>

**Expressions and Assignment**

A	<i>variable</i>
(/ 1.0, 2.0 /)	<i>array-constructor</i>
PERSON (12, 'Jones')	<i>structure-constructor</i>
F (X, Y)	<i>function-reference</i>
(S + T)	<i>(expr)</i>

6 **7.1.1.2 Level-1 Expressions.** Defined unary operators have the highest operator precedence  
 7 (Table 7.7). Level-1 expressions are primaries optionally operated on by defined unary operators:

- 8 R703 *level-1-expr* is [ *defined-unary-op* ] *primary*  
 9 R704 *defined-unary-op* is .*letter* [ *letter* ] ...

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

12 Simple examples of a level-1 expression are:

Example	Syntactic Class
A	<i>primary</i> (R701)
.INVERSE. B	<i>level-1-expr</i> (R703)

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

18 .INVERSE. (A + B)

19 **7.1.1.3 Level-2 Expressions.** Level-2 expressions are level-1 expressions optionally involving the  
 20 numeric operators *power-op*, *mult-op*, and *add-op*.

- 21 R705 *mult-operand* is *level-1-expr* [ *power-op mult-operand* ]  
 22 R706 *add-operand* is [ *add-operand mult-op* ] *mult-operand*  
 23 R707 *level-2-expr* is [ [ *level-2-expr* ] *add-op* ] *add-operand*  
 24 R708 *power-op* is \*\*  
 25 R709 *mult-op* is \*  
 26 or /  
 27 R710 *add-op* is +  
 28 or -

29 Simple examples of a level-2 expression are:

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

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

43 - A + D \* E + B \*\* C

1 **7.1.1.4 Level-3 Expressions.** Level-3 expressions are level-2 expressions optionally involving the  
 2 character operator *concat-op*.

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

4 R712 *concat-op* is //

5 Simple examples of a level-3 expression are:

Example	Syntactic Class
A	<i>level-2-expr</i> (R707)
B // C	<i>level-3-expr</i> (R711)

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

11 X // Y // 'ABCD'

12 **7.1.1.5 Level-4 Expressions.** Level-4 expressions are level-3 expressions optionally involving the  
 13 relational operators *rel-op*.

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

15 R714 *rel-op* is .EQ.

16 or .NE.

17 or .LT.

18 or .LE.

19 or .GT.

20 or .GE.

21 or ==

22 or /=

23 or < .

24 or <=

25 or >

26 or >=

27 Simple examples of a level-4 expression are:

Example	Syntactic Class
A	<i>level-3-expr</i> (R711)
B .EQ. C	<i>level-4-expr</i> (R713)
D < E	<i>level-4-expr</i> (R713)

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

34 (A + B) .NE. C

35 **7.1.1.6 Level-5 Expressions.** Level-5 expressions are level-4 expressions optionally involving the  
 36 logical operators *not-op*, *and-op*, *or-op*, and *equiv-op*.

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

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

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

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

41 R719 *not-op* is .NOT.

42 R720 *and-op* is .AND.

43 R721 *or-op* is .OR.

44 R722 *equiv-op* is .EQV.

## Expressions and Assignment

1 or .NEQV.

2 Simple examples of a level-5 expression are:

Example	Syntactic Class
A	<i>level-4-expr</i> (R713)
.NOT. B	<i>and-operand</i> (R715)
C .AND. D	<i>or-operand</i> (R716)
E .OR. F	<i>equi-operand</i> (R717)
G .EQV. H	<i>level-5-expr</i> (R718)
S .NEQV. T	<i>level-5-expr</i> (R718)

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

12 A .AND. B .EQV. .NOT. C

13 **7.1.1.7 General Form of an Expression.** Expressions are level-5 expressions optionally involving defined binary operators. Defined binary operators have the lowest operator precedence (Table 7.7).

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

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

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

20 Simple examples of an expression are:

Example	Syntactic Class
A	<i>level-5-expr</i> (R718)
B .UNION. C	<i>expr</i> (R723)

25 More complicated examples of an expression are:

26 (B .INTERSECT. C) .UNION. (X - Y)

27 A + B .EQ. C \* D

28 .INVERSE. (A + B)

29 A + B .AND. C \* D

30 E // G .EQ. H (1:10)

31 **7.1.2 Intrinsic Operations.** An *intrinsic operation* is either an *intrinsic unary operation* or an  
32 *intrinsic binary operation*. An *intrinsic unary operation* is an operation of the form *intrinsic-*  
33 *operator*  $x_2$  where  $x_2$  is of an *intrinsic type* (4.3) listed in Table 7.1 for the *unary intrinsic operator*.

34 An *intrinsic binary operation* is an operation of the form  $x_1$  *intrinsic-operator*  $x_2$  where  $x_1$  and  $x_2$   
35 are of the *intrinsic types* (4.3) listed in Table 7.1 for the *binary intrinsic operator* and are in shape  
36 conformance (7.1.5).

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

40 For *numeric intrinsic binary operations*, the two operands may be of different *numeric types* or  
41 different *type parameters*. Except for a value raised to an *integer power*, if the operands have dif-  
42 ferent types or *type parameters*, the effect is as if each operand that differs in type or *type parame-*  
43 *ter* from those of the result is converted to the type and *type parameter* of the result before the  
44 operation is performed. When a value of type *real* or *complex* is raised to an *integer power*, the  
45 *integer operand* need not be converted.



1 A character intrinsic operation, relational intrinsic operation, and logical intrinsic operation are  
 2 similarly defined in terms of a *character intrinsic operator* (`//`), *relational intrinsic operator* (`.EQ.`, `.NE.`,  
 3 `.GT.`, `.GE.`, `.LT.`, `.LE.`, `==`, `/=`, `>`, `>=`, `<`, and `<=`), and *logical intrinsic operator* (`.AND.`, `.OR.`, `.NOT.`,  
 4 `.EQV.`, and `.NEQV.`), respectively. For the intrinsic operator `//`, the kind type parameters must be  
 5 the same.

6 A numeric relational intrinsic operation is a relational intrinsic operation where the operands are  
 7 of numeric type. A character relational intrinsic operation is a relational intrinsic operation  
 8 where the operands are of type character and have the same kind type parameter value.

9 Table 7.1 Type of Operands and Result for the Intrinsic Operation  $[x_1] \text{ op } x_2$

10 <sup>Notes</sup> The symbols I, R, Z, C, and L stand for the types integer, real, complex, character, and logical,  
 11 respectively. Where more than one type for  $x_2$  is given, the type of the result of the operation is  
 12 given in the same relative position in the next column. For the intrinsic operators requiring oper-  
 13 ands of type character, the kind type parameters of the operands must be the same.

Intrinsic Operator <i>op</i>	Type of $x_1$	Type of $x_2$	Type of $[x_1] \text{ op } x_2$
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
<code>//</code>	C	C	C
<code>.EQ.</code> , <code>.NE.</code> , <code>==</code> , <code>/=</code>	I	I, R, Z	L, L, L
	R	I, R, Z	L, L, L
	Z	I, R, Z	L, L, L
	C	C	L
<code>.GT.</code> , <code>.GE.</code> , <code>.LT.</code> , <code>.LE.</code> <code>&gt;</code> , <code>&gt;=</code> , <code>&lt;</code> , <code>&lt;=</code>	I	I, R	L, L
	R	I, R	L, L
	C	C	L
<code>.NOT.</code>		L	L
<code>.AND.</code> , <code>.OR.</code> , <code>.EQV.</code> , <code>.NEQV.</code>	L	L	L

41 **7.1.3 Defined Operations.** A defined operation is either a defined unary operation or a defined  
 42 binary operation. A defined unary operation is an operation that has the form *defined-unary-op*  $x_2$   
 43 and that is defined by a function and a generic interface block (12.3.1) ~~that specifies the operation~~  
 44 ~~(7.3) for the operator *defined-unary-op*~~, or that has the form *intrinsic-operator*  $x_2$  where the type of  $x_2$   
 45 is not that required for a unary intrinsic operation (7.1.2), and that is defined by a function and a  
 46 generic interface block ~~that specifies the operation for the operator *intrinsic-operator*~~.

47 A defined binary operation is an operation that has the form  $x_1$  *defined-binary-op*  $x_2$  and that is  
 48 defined by a function and a generic interface block ~~that specifies the operation (7.3) for the opera-~~  
 49 ~~tor *defined-binary-op*~~, or that has the form  $x_1$  *intrinsic-operator*  $x_2$  where the types or ranks of either  
 50  $x_1$  or  $x_2$  or both are not those required for an intrinsic binary operation (7.1.2), and that is defined  
 51 by a function and a generic interface block ~~that specifies the operation for the operator *intrinsic-*~~  
 52 ~~operator~~.

53 Note that an intrinsic operator may be used as the operator in a defined operation. In such a case,  
 54 the generic properties of the operator are extended.

55 An extension operation is a defined operation in which the operator is of the form *defined-unary-op*  
 56 or *defined-binary-op*. Such an operator is called an extension operator. The operator used in an  
 57 extension operation may be such that a generic interface for the operator may specify more than

## Expressions and Assignment

1 one function.

2 **7.1.4 Data Type, Type Parameters, and Shape of an Expression.** The data type and shape of  
3 an expression depend on the operators and on the data types and shapes of the primaries used in  
4 the expression, and are determined recursively from the syntactic form of the expression. The data  
5 type of an expression is one of the intrinsic types (4.3) or a derived type (4.4).

6 R725 *logical-expr* is *expr*

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

8 R726 *char-expr* is *expr*

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

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

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

12 R728 *int-expr* is *expr*

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

14 R729 *numeric-expr* is *expr*

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

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

19 **7.1.4.1 Data Type, Type Parameters, and Shape of a Primary.** The data type, type parame-  
20 ters, and shape of a primary are determined according to whether the primary is a constant, vari-  
21 able, array constructor, structure constructor, function reference, or parenthesized expression. If a  
22 primary is a constant, its type, type parameters, and shape are those of the constant (4.3). If it is a  
23 structure constructor, it is scalar and its type is determined by the constructor name. If it is an  
24 array constructor, its type, type parameters, and shape are as described in 4.5. If it is a variable or  
25 function reference, its type, type parameters, and shape are those of the variable (5.1.1, 5.1.2) or the  
26 function reference (12.4.2), respectively. Note that in the case of a function reference, the function  
27 may be generic (12.3.2.1, 13.10), in which case its type, type parameters, and rank are determined  
28 by the types, type parameters, and ranks of its actual arguments. If a primary is a parenthesized  
29 expression, its type, type parameters, and shape are those of the expression.

30 If a pointer appears as a primary in an intrinsic operation or a defined operation in which it corre-  
31 sponds to a nonpointer dummy argument, the associated target object is referenced. The type,  
32 type parameters, and shape of the primary are those of the current target. If the pointer is not  
33 associated with a target, it may appear as a primary only as an actual argument in a reference to a  
34 procedure whose corresponding dummy argument is declared to be a pointer.

35 **7.1.4.2 Data Type, Type Parameters, and Shape of the Result of an Operation.** The type of  
36 the result of an intrinsic operation  $[x_1] \text{ op } x_2$  is specified by Table 7.1. The type of the result of a  
37 defined operation  $[x_1] \text{ op } x_2$  is specified by the function defining the operation (7.3).

38 The shape of the result of an intrinsic operation is the shape of  $x_2$  if *op* is unary or if  $x_1$  is scalar,  
39 and is the shape of  $x_1$  otherwise.

40 An expression of an intrinsic type has a kind type parameter. An expression of type character also  
41 has a length type parameter. For an expression  $x_1 // x_2$  where  $x_1$  and  $x_2$  are of type character, the  
42 length type parameter is the sum of the lengths of the operands and the kind type parameter is the  
43 kind type parameter of  $x_1$ , which must be the same as the kind type parameter of  $x_2$ . For an  
44 expression  $\text{op } x_2$  where *op* is an intrinsic unary operator and  $x_2$  is of type integer, real, complex, or  
45 logical, the kind type parameter of the expression is that of the operand. For an expression  $x_1 \text{ op } x_2$   
46 where *op* is a numeric intrinsic binary operator with one operand of type integer and the other of  
47 type real or complex, the kind type parameter of the expression is that of the real or complex

1 operand. For an expression  $x_1 \text{ op } x_2$  where *op* is a numeric intrinsic binary operator with both  
 2 operands of the same type and kind type parameters, or with one real and one complex with the  
 3 same kind type parameters, the kind type parameter of the expression is identical to that of each  
 4 operand. In the case where both operands are integer with different kind type parameters, the  
 5 kind type parameter of the expression is that of the operand with the greater decimal exponent  
 6 range or is processor dependent if the operands have the same decimal exponent range. In the  
 7 case where both operands are any of type real or complex with different kind type parameters, the  
 8 kind type parameter of the expression is that of the operand with the greater decimal precision or  
 9 is processor dependent if the operands have the same decimal precision. For an expression  $x_1 \text{ op}$   
 10  $x_2$  where *op* is a logical intrinsic binary operator with both operands of the same kind type parame-  
 11 ter, the kind type parameter of the expression is identical to that of each operand. In the case  
 12 where both operands are of type logical with different kind type parameters, the kind type param-  
 13 eter of the expression is processor dependent. For an expression  $x_1 \text{ op } x_2$  where *op* is a relational  
 14 intrinsic operator, the expression has the default logical kind type parameter.

15 **7.1.5 Conformability Rules for Intrinsic Operations.** Two entities are in shape conformance if  
 16 both are arrays of the same shape, or one or both are scalars.

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

20 **7.1.6 Scalar and Array Expressions.** An expression is either a scalar expression or an array  
 21 expression.

22 The following is an example of a scalar expression:

23  $Q + 2.3 * R$

24 where Q and R are scalars.

25 The following is an example of an array expression:

26  $A (1:10) + B (2:11)$

27 where A and B are arrays.

28 **7.1.6.1 Constant Expression.** A constant expression is an expression in which each operation is  
 29 intrinsic and each primary is one of the following:

- 30 (1) A constant or subobject of a constant where each subscript, section subscript, substring  
 31 starting point, and substring ending point is a constant expression.
- 32 (2) An array constructor where each element and the bounds and strides of each implied-  
 33 DO are expressions whose primaries are either constant expressions or implied-DO var-  
 34 iables,
- 35 (3) A structure constructor where each component is a constant expression,
- 36 (4) An elemental intrinsic function reference where each argument is a constant expression,
- 37 (5) A transformational intrinsic function reference where each argument is a constant  
 38 expression,
- 39 (6) A reference to an array inquiry function (13.10.15) other than ALLOCATED, the bit  
 40 inquiry function BIT\_SIZE, the character inquiry function LEN, the kind inquiry func-  
 41 tion KIND, or a numeric inquiry function (13.10.8) where each argument is either a con-  
 42 stant expression or a variable whose type parameters or bounds inquired about are not  
 43 assumed or defined by an ALLOCATE statement or a pointer assignment, or
- 44 (7) A constant expression enclosed in parentheses.

45 A character constant expression is a constant expression whose type is character. An integer con-  
 46 stant expression is a constant expression whose type is integer. A logical constant expression is a  
 47 constant expression whose type is logical. A numeric constant expression is a constant expression

## Expressions and Assignment

- 1 whose type is integer, real, or complex.
- 2 An initialization expression is a constant expression in which the exponentiation operation is per-  
3 mitted only with an integer power, and each primary is one of the following:
  - 4 (1) A constant or subobject of a constant where each subscript, section subscript, substring  
5 starting point, and substring ending point is an initialization expression,
  - 6 (2) An array constructor where each element and the bounds and strides of each implied-  
7 DO are expressions whose primaries are either initialization expressions or implied-DO  
8 variables,
  - 9 (3) A structure constructor where each component is an initialization expression,
  - 10 (4) An elemental intrinsic function reference of type integer or character where each argu-  
11 ment is an initialization expression of type integer or character,
  - 12 (5) A reference to one of the transformational functions, REPEAT, RESHAPE,  
13 SELECTED\_INT\_KIND, SELECTED\_REAL\_KIND, TRANSFER, or TRIM, where each  
14 argument is an initialization expression,
  - 15 (6) A reference to an array inquiry function (13.10.15) other than ALLOCATED, the bit  
16 inquiry function BIT\_SIZE, the character inquiry function LEN, the kind inquiry func-  
17 tion KIND, or a numeric inquiry function (13.10.8) where each argument is either an  
18 initialization expression or a variable whose type parameters or bounds inquired about  
19 are not assumed or defined by an ALLOCATE statement or a pointer assignment, or  
20 (7) An initialization expression enclosed in parentheses.

21	R730	<i>initialization-expr</i>	is <i>expr</i>
22	R731	<i>char-initialization-expr</i>	is <i>char-expr</i>
23	R732	<i>int-initialization-expr</i>	is <i>int-expr</i>
24	R733	<i>logical-initialization-expr</i>	is <i>logical-expr</i>

25 If an initialization expression includes a reference to an inquiry function for a type parameter or an  
26 array bound of an object specified in the same *specification-part*, the type parameter or array bound  
27 must be specified in a prior specification of the *specification-part*. The prior specification may be to  
28 the left of the inquiry function in the same statement.

29 The following are examples of constant expressions:

```
30 3  
31 -3 + 4  
32 'AB'  
33 'AB' // 'CD'  
34 ('AB' // 'CD') // 'EF'  
35 SIZE (A)  
36 DIGITS (X) + 4
```

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

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

```
39 ABS (9.0) ! Not an integer argument  
40 3.0 ** 2.0 ! Not an integer power  
41 DOT_PRODUCT ( (/ 2, 3 /), (/ 1, 7 /) ) ! Not an allowed function
```

42 **7.1.6.2 Specification Expression.** A restricted expression is an expression in which each opera-  
43 tion is intrinsic and each primary is:

- 44 (1) A constant or subobject of a constant ~~where each subscript, section subscript, substring~~  
45 ~~starting point, and substring ending point is a restricted expression,~~

- 1 (2) A variable that is a dummy argument; *or a variable that is a subobject of a dummy argument*
- 2 (3) A variable that is in a common block; *or a variable that is a subobject of a variable in a common block*
- 3 (4) A variable that is made accessible by use association or host association; *or a variable that is a subobject of such a variable.*
- 4 (5) An array constructor where each element and the bounds and strides of each implied-DO are expressions whose primaries are either restricted expressions or implied-DO variables,
- 5
- 6
- 7 (6) A structure constructor where each component is a restricted expression,
- 8 (7) An elemental intrinsic function reference of type integer or character where each argument is a restricted expression of type integer or character,
- 9
- 10 (8) One of the transformational functions REPEAT, RESHAPE, SELECTED\_INT\_KIND,
- 11 SELECTED\_REAL\_KIND, TRANSFER, and TRIM, where each argument is a restricted
- 12 expression of type integer or character,
- 13 (9) A reference to an array inquiry function (13.10.15) other than ALLOCATED, the bit
- 14 inquiry function BIT\_SIZE, the character inquiry function LEN, the kind inquiry func-
- 15 tion KIND, or a numeric inquiry function (13.10.8) where each argument is either a
- 16 restricted expression or a variable whose type parameters or bounds inquired about are
- 17 not assumed or defined by an ALLOCATE statement or a pointer assignment, or

18 (10) A restricted expression enclosed in parentheses, *and where any subobject, section subobject, subscripting starting point or subscripting ending point is a restricted expression.*

19 A specification expression (R509, R514, R515) is a restricted expression that is scalar and of type

20 integer.

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

22 Constraint: The *scalar-int-expr* must be a restricted expression.

23 A variable in a specification expression must have its type and type parameters, if any, specified

24 by a previous declaration in the same scoping unit, or by the implicit typing rules currently in

25 effect for the scoping unit, or by host or use association. If a variable in a specification expression

26 is typed by the implicit typing rules, ~~if~~ *its* appearance in any subsequent declaration statement must

27 confirm this implied type and type parameters. *type ✓ 116*

28 If a specification expression includes a reference to an inquiry function for a type parameter or an

29 array bound of an entity specified in the same *specification-part*, the type parameter or array bound

30 must be specified in a prior specification of the *specification-part*. If a specification expression

31 includes a reference to the value of an element of an array specified in the same *specification-part*,

32 the array bounds must be specified in a prior declaration. The prior specification may be to the left

33 of the inquiry function in the same statement.

34 The following are examples of specification expressions:

```

35 LBOUND (B, 1) + 5 ! B is an assumed-shape dummy array
36 M + LEN (C) ! M and C are dummy arguments
37 2 * PRECISION (A) ! A is a real variable made accessible
38 ! by a USE statement
    
```

39 **7.1.7 Evaluation of Operations.** This section applies to both intrinsic and defined operations.

40 Any variable or function reference used as an operand in an expression must be defined at the

41 time the reference is executed. If the operand is a pointer, it must be associated with a target object

42 that is defined. An integer operand must be defined with an integer value rather than a statement label

43 value. All of the characters in a character data object reference must be defined.

44 When a reference to an array or an array section is made, all of the selected elements must be

45 defined. When a structure is referenced, all of the components must be defined.

46 Any numeric operation whose result is not mathematically defined is prohibited in the execution

47 of an executable program. Examples are dividing by zero and raising a zero-valued primary to a

## Expressions and Assignment

1 zero-valued or negative-valued power. Raising a negative-valued primary of type real to a real  
2 power also is prohibited.

3 The evaluation of a function reference must neither affect nor be affected by the evaluation of any  
4 other entity within the statement. However, execution of a function reference in the logical expres-  
5 sion of an IF statement (8.1.2.4) or WHERE statement (7.5.3.1) is permitted to define variables in  
6 the statement that is executed when the value of the expression is true. For example, in the state-  
7 ments:

8 IF (F (X)) A = X  
9 WHERE (G (X) ) B = X

10 F or G may define X. If a function reference causes definition or undefinition of an actual argu-  
11 ment of the function, that argument or any associated entities must not appear elsewhere in the  
12 same statement. For example, the statements

13 A (I) = F (I)  
14 Y = G (X) + X

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

17 The type of an expression in which a function reference appears does not affect, and is not affected  
18 by, the evaluation of the actual arguments of the function.

19 Execution of an array element reference requires the evaluation of its subscripts. The type of an  
20 expression in which the array element reference appears does not affect, and is not affected by, the  
21 evaluation of its subscripts. Execution of an array section reference requires the evaluation of its  
22 section subscripts. The type of an expression in which an array section appears does not affect,  
23 and is not affected by, the evaluation of the array section subscripts. Execution of a substring refer-  
24 ence requires the evaluation of its substring expressions. The type of an expression in which a  
25 substring appears does not affect, and is not affected by, the evaluation of the substring expres-  
26 sions. It is not necessary for a processor to evaluate any subscript expressions or substring expres-  
27 sions for an array of zero size or a character entity of zero length.

28 The appearance of an array constructor requires the evaluation of the bounds and stride of any  
29 array constructor implied-DO it may contain. The type of an expression in which an array con-  
30 structor appears does not affect, and is not affected by, evaluation of such bounds and stride  
31 expressions.

32 When an intrinsic binary operation is applied to a scalar and an array or to two arrays of the same  
33 shape, the operation is performed element-by-element on corresponding array elements of the  
34 array operands. For example, the array expression

35 A + B

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

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

42 **7.1.7.1 Evaluation of Operands.** It is not necessary for a processor to evaluate all of the oper-  
43 ands of an expression if the value of the expression can be determined otherwise. This principle is  
44 most often applicable to logical expressions, zero-sized arrays, and zero-length strings, but it  
45 applies to all expressions. For example, in evaluating the expression

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

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

1  $W(Z) + X$

2 where  $X$  is of size zero and  $W$  is a function, the function reference  $W(Z)$  need not be evaluated. If  
 3 a statement contains a function reference in a part of an expression that need not be evaluated, all  
 4 entities that would have become defined in the execution of that reference become undefined at  
 5 the completion of evaluation of the expression containing the function reference. In the preceding  
 6 examples, evaluation of these expressions causes  $Z$  to become undefined if  $L$  or  $W$  defines its argu-  
 7 ment.

8 **7.1.7.2 Integrity of Parentheses.** The sections that follow state certain conditions under which a  
 9 processor may evaluate an expression that is different from the one specified by applying the rules  
 10 given in 7.1.1, 7.2, and 7.3. However, any expression contained in parentheses must be treated as a  
 11 data entity. For example, in evaluating the expression  $A + (B - C)$  where  $A$ ,  $B$ , and  $C$  are of  
 12 numeric types, the difference of  $B$  and  $C$  must be evaluated before the addition operation is per-  
 13 formed; the processor must not evaluate the mathematically equivalent expression  $(A + B) - C$ .

14 **7.1.7.3 Evaluation of Numeric Intrinsic Operations.** The rules given in 7.2.1 specify the inter-  
 15 pretation of a numeric intrinsic operation. Once the interpretation has been established in accord-  
 16 ance with those rules, the processor may evaluate any mathematically equivalent expression, pro-  
 17 vided that the integrity of parentheses is not violated.

18 Two expressions of a numeric type are mathematically equivalent if, for all possible values of their  
 19 primaries, their mathematical values are equal. However, mathematically equivalent expressions  
 20 of numeric type may produce different computational results. For example, any difference  
 21 between the values of the expressions  $(1./3.)*3.$  and  $1.$  is a computational difference, not a mathe-  
 22 matical difference.

23 The mathematical definition of integer division is given in 7.2.1.1. The difference between the val-  
 24 ues of the expressions  $5/2$  and  $5./2.$  is a mathematical difference, not a computational difference.

25 The following are examples of expressions with allowable alternative forms that may be used by  
 26 the processor in the evaluation of those expressions.  $A$ ,  $B$ , and  $C$  represent arbitrary real or com-  
 27 plex operands;  $I$  and  $J$  represent arbitrary integer operands; and  $X$ ,  $Y$ , and  $Z$  represent arbitrary  
 28 operands of numeric type.

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

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

Expression	Nonallowable Alternative Form
$I / 2$	$0.5 * I$
$X * I / J$	$X * (I / J)$
$I / J / A$	$I / (J * A)$
$(X + Y) + Z$	$X + (Y + Z)$
$(X * Y) - (X * Z)$	$X * (Y - Z)$
$X * (Y - Z)$	$X * Y - X * Z$

## Expressions and Assignment

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

5  $A + (B - C)$

6 the parenthesized expression  $(B - C)$  must be evaluated and then added to  $A$ .

7 Note that the inclusion of parentheses may change the mathematical value of an expression. For  
8 example, the two expressions:

9  $A * I / J$

10  $A * (I / J)$

11 may have different mathematical values if  $I$  and  $J$  are of type integer.

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

14  $Z + R + I$

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

18 **7.1.7.4 Evaluation of the Character Intrinsic Operation.** The rules given in 7.2.2 specify the  
19 interpretation of a character intrinsic operation. A processor needs to evaluate only as much of the  
20 character intrinsic operation as is required by the context in which the expression appears. For  
21 example, the statements

22 CHARACTER (LEN = 2) C1, C2, C3, CF

23 C1 = C2 // CF (C3)

24 do not require the function  $CF$  to be evaluated, because only the value of  $C2$  is needed to deter-  
25 mine the value of  $C1$  because  $C1$  has a length of 2.

26 **7.1.7.5 Evaluation of Relational Intrinsic Operations.** The rules given in 7.2.3 specify the inter-  
27 pretation of relational intrinsic operations. Once the interpretation of an expression has been  
28 established in accordance with those rules, the processor may evaluate any other expression that is  
29 relationally equivalent, provided that the integrity of parentheses in any expression is not violated.  
30 For example, the processor may choose to evaluate the expression

31  $I .GT. J$

32 where  $I$  and  $J$  are integer variables, as

33  $J - I .LT. 0$

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

36 **7.1.7.6 Evaluation of Logical Intrinsic Operations.** The rules given in 7.2.4 specify the interpre-  
37 tation of logical intrinsic operations. Once the interpretation of an expression has been established  
38 in accordance with those rules, the processor may evaluate any other expression that is logically  
39 equivalent, provided that the integrity of parentheses in any expression is not violated. For exam-  
40 ple, for the variables  $L1$ ,  $L2$ , and  $L3$  of type logical, the processor may choose to evaluate the  
41 expression

42  $L1 .AND. L2 .AND. L3$

43 as

44  $L1 .AND. (L2 .AND. L3)$



- 1 Two expressions of type logical are logically equivalent if their values are equal for all possible values of their primaries.  
2
- 3 **7.1.7.7 Evaluation of a Defined Operation.** The rules given in 7.3 specify the interpretation of a  
4 defined operation. Once the interpretation of an expression has been established in accordance  
5 with those rules, the processor may evaluate any other expression that is equivalent, provided that  
6 the integrity of parentheses is not violated.
- 7 Two expressions of derived type are equivalent if their values are equal for all possible values of  
8 their primaries.

9 **7.2 Interpretation of Intrinsic Operations.** The intrinsic operations are those defined in  
10 7.1.2. These operations are divided into the following categories: numeric, character, relational,  
11 and logical. The interpretations defined in the following sections apply to both scalars and arrays;  
12 the interpretation for arrays is obtained by applying the interpretation for scalars element by element.  
13

14 The type, type parameters, and interpretation of an expression that consists of an intrinsic unary or  
15 binary operation are independent of the context in which the expression appears. In particular, the  
16 type, type parameters, and interpretation of such an expression are independent of the type and  
17 type parameters of any other larger expression in which it appears. For example, if  $X$  is of type  
18 real,  $J$  is of type integer, and  $INT$  is the real-to-integer intrinsic conversion function, the expression  
19  $INT(X + J)$  is an integer expression and  $X + J$  is a real expression.

20 **7.2.1 Numeric Intrinsic Operations.** A numeric operation is used to express a numeric computation.  
21 Evaluation of a numeric operation produces a numeric value. The permitted data types for  
22 operands of the numeric intrinsic operations are specified in 7.1.2.

23 The numeric operators and their interpretation in an expression are given in Table 7.2, where  $x_1$   
24 denotes the operand to the left of the operator and  $x_2$  denotes the operand to the right of the operator.  
25

26 **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 $x_2$

- 41 The interpretation of a division depends on the data types of the operands (7.2.1.1).  
42 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  
43 same as the interpretation of  $1/(x_1 ** ABS(x_2))$ , which is subject to the rules of integer division  
44 (7.2.1.1). For example,  $2 ** (-3)$  has the value of  $1/(2 ** 3)$ , which is zero.
- 45 **7.2.1.1 Integer Division.** One operand of type integer may be divided by another operand of  
46 type integer. Although the mathematical quotient of two integers is not necessarily an integer,  
47 Table 7.1 specifies that an expression involving the division operator with two operands of type  
48 integer is interpreted as an expression of type integer. The result of such an operation is the integer  
49 closest to the mathematical quotient and between zero and the mathematical quotient inclusively.  
50 For example, the expression  $(-8) / 3$  has the value  $(-2)$ .

**Expressions and Assignment**

- 1 **7.2.1.2 Complex Exponentiation.** In the case of a complex value raised to a complex power, the
- 2 value of the operation  $x_1 ** x_2$  is the principal value of  $x_1^{x_2}$ .
- 3 **7.2.2 Character Intrinsic Operation.** The character intrinsic operator // is used to concatenate
- 4 two operands of type character with the same kind type parameter. Evaluation of the character
- 5 intrinsic operation produces a result of type character.
- 6 The interpretation of the character intrinsic operator // when used to form an expression is given
- 7 in Table 7.3, where  $x_1$  denotes the operand to the left of the operator and  $x_2$  denotes the operand to
- 8 the right of the operator.

9 **Table 7.3** Interpretation of the Character Intrinsic Operator //

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

18 The result of a character intrinsic operation is a character string whose value is the value of  $x_1$  con-

19 catenated on the right with the value of  $x_2$  and whose length is the sum of the lengths of  $x_1$  and  $x_2$ .

20 Parentheses used to specify the order of evaluation have no effect on the value of a character

21 expression. For example, the value of ('AB' // 'CDE') // 'F' is the string 'ABCDEF'. Also, the

22 value of 'AB' // ('CDE' // 'F') is the string 'ABCDEF'.

23 **7.2.3 Relational Intrinsic Operations.** A relational intrinsic operator is used to compare values

24 of two operands using the relational intrinsic operators .LT., .LE., .GT., .GE., .EQ., .NE., <, <=, >,

25 >=, ==, and /=. The permitted data types for operands of the relational intrinsic operators are

26 specified in 7.1.2. Note, as shown in Table 7.1, that a relational intrinsic operator must not be used

27 to compare the value of an expression of a numeric type with one of type character or logical.

28 Also, two operands of type logical must not be compared, a complex operand may be compared

29 with another numeric operand only when the operator is .EQ., .NE., ==, or /=, and two character

30 operands must not be compared unless they have the same kind type parameter value.

31 Evaluation of a relational intrinsic operation produces a result of type default logical.

32 The interpretation of the relational intrinsic operators is given in Table 7.4, where  $x_1$  denotes the

33 operand to the left of the operator and  $x_2$  denotes the operand to the right of the operator. The

34 operators <, <=, >, >=, ==, and /= always have the same interpretations as the operators .LT., .LE.,

35 .GT., .GE., .EQ., and .NE., respectively.

36 **Table 7.4** Interpretation of the Relational Intrinsic Operators

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

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

5 In the numeric relational operation

6  $x_1 \text{ rel-op } x_2$

7 if the type or type parameters of  $x_1$  and  $x_2$  differ, their values are converted to the type and type  
 8 parameters of the expression  $x_1 + x_2$  before evaluation.

9 A character relational intrinsic operation is interpreted as having the logical value true if the val-  
 10 ues of the operands satisfy the relation specified by the operator. A character relational intrinsic  
 11 operation is interpreted as having the logical value false if the values of the operands do not satisfy  
 12 the relation specified by the operator.

13 For a character relational intrinsic operation, the operands are compared one character at a time in  
 14 order, beginning with the first character of each character operand. If the operands are of unequal  
 15 length, the shorter operand is treated as if it were extended on the right with blanks to the length  
 16 of the longer operand. If both  $x_1$  and  $x_2$  are of zero length,  $x_1$  is equal to  $x_2$ ; if every character of  $x_1$   
 17 is the same as the character in the corresponding position in  $x_2$ ,  $x_1$  is equal to  $x_2$ . Otherwise, at the  
 18 first position where the character operands differ, the character operand  $x_1$  is considered to be less  
 19 than  $x_2$  if the character value of  $x_1$  at this position precedes the value of  $x_2$  in the collating  
 20 sequence (3.1.7);  $x_1$  is greater than  $x_2$  if the character value of  $x_1$  at this position follows the value  
 21 of  $x_2$  in the collating sequence. Note that the collating sequence depends partially on the proces-  
 22 sor; however, the result of the use of the operators .EQ., .NE., ==, and /= does not depend on the  
 23 collating sequence.

24 Note that for nondefault character types, the blank padding character is processor dependent.

25 **7.2.4 Logical Intrinsic Operations.** A logical operation is used to express a logical computation.  
 26 Evaluation of a logical operation produces a result of type logical. The permitted data types for  
 27 operands of the logical intrinsic operations are specified in 7.1.2.

28 The logical operators and their interpretation when used to form an expression are given in Table  
 29 7.5, where  $x_1$  denotes the operand to the left of the operator and  $x_2$  denotes the operand to the  
 30 right of the operator.

31 Table 7.5 Interpretation of the Logical Intrinsic Operators

Operator	Representing	Use of Operator	Interpretation
.NOT.	Logical Negation	.NOT. $x_2$	True if $x_2$ is false
.AND.	Logical Conjunction	$x_1$ .AND. $x_2$	True if $x_1$ and $x_2$ are both true
.OR.	Logical Inclusive Disjunction	$x_1$ .OR. $x_2$	True if $x_1$ and/or $x_2$ is true
.NEQV.	Logical Non-equivalence	$x_1$ .NEQV. $x_2$	True if either $x_1$ or $x_2$ is true, but not both
.EQV.	Logical Equivalence	$x_1$ .EQV. $x_2$	True if both $x_1$ and $x_2$ are true or both are false

44 The values of the logical intrinsic operations are shown in Table 7.6.

45 Table 7.6 The Values of Operations Involving Logical Intrinsic Operators

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

1 **7.3 Interpretation of Defined Operations.** The interpretation of a defined operation is pro-  
 2 vided by the function that defines the operation. The type, type parameters, and interpretation of  
 3 an expression that consists of a defined operation are independent of the type and type parameters  
 4 of any larger expression in which it appears. The operators `<`, `<=`, `>`, `>=`, `==`, and `/=` always have  
 5 the same interpretations as the operators `.LT.`, `.LE.`, `.GT.`, `.GE.`, `.EQ.`, and `.NE.`, respectively.

6 **7.3.1 Unary Defined Operation.** A function defines the unary operation  $op\ x_2$  if:

- 7 (1) The function is specified with a FUNCTION (12.5.2.2) or ENTRY (12.5.2.5) statement
- 8 that specifies one dummy argument  $d_2$ ,
- 9 (2) An interface block (12.3.2.1) provides the function with a *generic-spec* of OPERATOR
- 10 ( $op$ ),
- 11 (3) The type of  $x_2$  is the same as the type of dummy argument  $d_2$ ,
- 12 (4) The type parameters, if any, of  $x_2$  match those of  $d_2$ ,
- 13 (5) The rank of  $x_2$ , and its shape if it is an array, match those of  $d_2$ .

14 **7.3.2 Binary Defined Operation.** A function defines the binary operation  $x_1\ op\ x_2$  if:

- 15 (1) The function is specified with a FUNCTION (12.5.2.2) or ENTRY (12.5.2.5) statement
- 16 that specifies two dummy arguments,  $d_1$  and  $d_2$ ,
- 17 (2) An interface block (12.3.2.1) provides the function with a *generic-spec* of OPERATOR
- 18 ( $op$ ),
- 19 (3) The types of  $x_1$  and  $x_2$  are the same as those of the dummy arguments  $d_1$  and  $d_2$ , respec-
- 20 tively,
- 21 (4) The type parameters, if any, of  $x_1$  and  $x_2$  match those of  $d_1$  and  $d_2$ , respectively, and
- 22 (5) The ranks of  $x_1$  and  $x_2$ , and their shapes if either or both are arrays, match those of  $d_1$
- 23 and  $d_2$ , respectively.

24 **7.4 Precedence of Operators.** There is a precedence among the intrinsic and extension  
 25 operations implied by the general form in 7.1.1, which determines the order in which the operands  
 26 are combined, unless the order is changed by the use of parentheses. This precedence order is  
 27 summarized in Table 7.7.

28 **Table 7.7** Categories of Operations and Relative Precedences

Category of Operation	Operators	Precedence
Extension	<i>defined-unary-op</i>	Highest
Numeric	<code>**</code>	.
Numeric	<code>*</code> or <code>/</code>	.
Numeric	unary <code>+</code> or <code>-</code>	.
Numeric	binary <code>+</code> or <code>-</code>	.
Character	<code>//</code>	.
Relational	<code>.EQ.</code> , <code>.NE.</code> , <code>.LT.</code> , <code>.LE.</code> , <code>.GT.</code> , <code>.GE.</code>	.
	<code>==</code> , <code>/=</code> , <code>&lt;</code> , <code>&lt;=</code> , <code>&gt;</code> , <code>&gt;=</code>	.
Logical	<code>.NOT.</code>	.
Logical	<code>.AND.</code>	.
Logical	<code>.OR.</code>	.
Logical	<code>.EQV.</code> or <code>.NEQV.</code>	.
Extension	<i>defined-binary-op</i>	Lowest

1 The precedence of a defined operation is that of its operator.

2 For example, in the expression

3  $-A ** 2$

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

8  $-(A ** 2)$

9 The general form of an expression (7.1.1) also establishes a precedence among operators in the  
10 same syntactic class. This precedence determines the order in which the operands are to be com-  
11 bined in determining the interpretation of the expression unless the order is changed by the use of  
12 parentheses. For example, in interpreting a *level-2-expr* containing two or more binary operators +  
13 or -, each operand (*add-operand*) is combined from left to right. Similarly, the same left-to-right  
14 interpretation for a *mult-operand* in *add-operand*, as well as for other kinds of expressions, is a conse-  
15 quence of the general form. However, for interpreting a *mult-operand* expression when two or  
16 more exponentiation operators  $**$  combine *level-1-expr* operands, each *level-1-expr* is combined  
17 from right to left. For example, the expressions

18  $2.1 + 3.4 + 4.9$

19  $2.1 * 3.4 * 4.9$

20  $2.1 / 3.4 / 4.9$

21  $2 ** 3 ** 4$

22  $'AB' // 'CD' // 'EF'$

23 have the same interpretations as the expressions

24  $(2.1 + 3.4) + 4.9$

25  $(2.1 * 3.4) * 4.9$

26  $(2.1 / 3.4) / 4.9$

27  $2 ** (3 ** 4)$

28  $('AB' // 'CD') // 'EF'$

29 As a consequence of the general form (7.1.1), only the first *add-operand* of a *level-2-expr* may be pre-  
30 ceded by the identity (+) or negation (-) operator. These formation rules do not permit expres-  
31 sions containing two consecutive numeric operators, such as  $A ** -B$  or  $A + -B$ . However, expres-  
32 sions such as  $A ** (-B)$  and  $A + (-B)$  are permitted. The rules do allow a binary operator or an  
33 intrinsic unary operator to be followed by a defined unary operator, such as:

34  $A * .INVERSE. B$

35  $- .INVERSE. (B)$

36 As another example, in the expression

37  $A .OR. B .AND. C$

38 the general form implies a higher precedence for the  $.AND.$  operator than for the  $.OR.$  operator;  
39 therefore, the interpretation of the above expression is the same as the interpretation of the expres-  
40 sion

41  $A .OR. (B .AND. C)$

42 An expression may contain more than one category of operator. For example, the logical expres-  
43 sion

44  $L .OR. A + B .GE. C$

45 where A, B, and C are of type real, and L is of type logical, contains a numeric operator, a rela-  
46 tional operator, and a logical operator. This expression would be interpreted the same as the  
47 expression

48  $L .OR. ((A + B) .GE. C)$

## Expressions and Assignment

1 For example, if:

2 (1) The operator **\*\*** is extended to type logical,

3 (2) The operator **.STARSTAR.** is defined to duplicate the function of **\*\*** on type real,

4 (3) **.MINUS.** is defined to duplicate the unary operator **-**, and

5 (4) L1 and L2 are type logical and X and Y are type real,

6 then in precedence: L1 **\*\*** L2 is higher than X \* Y; X \* Y is higher than X **.STARSTAR.** Y; and

7 **.MINUS.** X is higher than **-X**.

8 **7.5 Assignment.** Execution of an assignment statement causes a variable to become defined or  
9 redefined. Execution of a pointer assignment statement causes a pointer to become associated  
10 with a target or causes its pointer association status to become associated or undefined. Execution  
11 of a WHERE statement or WHERE construct masks the evaluation of expressions and assignment  
12 of values in array assignment statements according to the value of a logical array expression.

13 **7.5.1 Assignment Statement.** A variable may be defined or redefined by execution of an assign-  
14 ment statement.

### 15 7.5.1.1 General Form.

16 R735 *assignment-stmt* is *variable = expr*

17 where *variable* is defined in R601 and *expr* is defined in R723.

18 Constraint: A *variable* in an *assignment-stmt* must not be an assumed-size array.

19 Examples of an assignment statement are:

20 A = 3.5 + X \* Y

21 I = INT (A)

22 An assignment statement is either intrinsic or defined.

23 **7.5.1.2 Intrinsic Assignment Statement.** An intrinsic assignment statement is an assignment  
24 statement where the shapes of *variable* and *expr* conform and where:

25 (1) The types of *variable* and *expr* are intrinsic, as specified in Table 7.8 for assignment, or

26 (2) The types of *variable* and *expr* are of the same derived type and no defined assignment  
27 exists for objects of this type.

28 A numeric intrinsic assignment statement is an intrinsic assignment statement for which *variable*  
29 and *expr* are of numeric type. A character intrinsic assignment statement is an intrinsic assign-  
30 ment statement for which *variable* and *expr* are of type character and have the same kind type  
31 parameter. A logical intrinsic assignment statement is an intrinsic assignment statement for  
32 which *variable* and *expr* are of type logical. A derived-type intrinsic assignment statement is an  
33 intrinsic assignment statement for which *variable* and *expr* are of the same derived type, and there is  
no interface to such with an ASSIGNMENT (or) specifier for objects of the derived type.  
34 An array intrinsic assignment statement is an intrinsic assignment statement for which *variable* is  
35 an array. The *variable* must not be a many-one array section (6.2.2.3.2).

36 Table 7.8 Type Conformance for the Intrinsic Assignment Statement *variable = expr*

Type of <i>variable</i>	Type of <i>expr</i>
integer	integer, real, complex
real	integer, real, complex
complex	integer, real, complex
character	character of the same kind type parameter as <i>variable</i>
logical	logical
derived type	same derived type as <i>variable</i>

13 **7.5.1.3 Defined Assignment Statement.** A defined assignment statement is an assignment  
 14 statement that is not an intrinsic assignment statement, and is defined by a subroutine ~~that has an~~ *and an*  
 15 ~~explicit ASSIGNMENT(=) interface (12.3.2.1).~~ *that specifies ASSIGNMENT(=).*

16 **7.5.1.4 Intrinsic Assignment Conformance Rules.** For an intrinsic assignment statement, *vari-*  
 17 *able* and *expr* must conform in shape, and if *expr* is an array, *variable* must also be an array. The  
 18 types of *variable* and *expr* must conform with the rules of Table 7.8.

19 If *variable* is a pointer, it must be associated with a definable target such that the type, type paramete-  
 20 ters, and shape of the target and *expr* conform.

21 For a numeric intrinsic assignment statement, *variable* and *expr* may have different numeric types  
 22 or different type parameters, in which case the value of *expr* is converted to the type and type  
 23 parameters of *variable* according to the rules of Table 7.9.

24 **Table 7.9** Numeric Conversion and Assignment Statement *variable = expr*

Type of <i>variable</i>	Value Assigned
integer	INT ( <i>expr</i> , KIND = KIND ( <i>variable</i> ))
real	REAL ( <i>expr</i> , KIND = KIND ( <i>variable</i> ))
complex	CMPLX ( <i>expr</i> , KIND = KIND ( <i>variable</i> ))

36 (The functions INT, REAL, CMPLX, and KIND are the generic functions defined in 13.13.)

37 For a character intrinsic assignment statement, *variable* and *expr* must have the same kind type  
 38 parameter value, but may have different length type parameters in which case the conversion of  
 39 *expr* to the length of *variable* is:

- 40 (1) If the length of *variable* is less than that of *expr*, the value of *expr* is truncated from the  
 41 right until it is the same length as *variable*;
- 42 (2) If the length of *variable* is greater than that of *expr*, the value of *expr* is extended on the  
 43 right with blanks until it is the same length as *variable*.

44 Note that for nondefault character types, the blank padding character is processor dependent.

45 **7.5.1.5 Interpretation of Intrinsic Assignments.** Execution of an intrinsic assignment causes, in  
 46 effect, the evaluation of the expression *expr* and all expressions within *variable* (7.1.7), the possible  
 47 conversion of *expr* to the type and type parameters of *variable* (Table 7.9), and the definition of *vari-*  
 48 *able* with the resulting value. The execution of the assignment must have the same effect as if the  
 49 evaluation of all operations in *expr* and *variable* occurred before any portion of *variable* is defined by  
 50 the assignment. The evaluation of expressions within *variable* must neither affect nor be affected  
 51 by the evaluation of *expr*. No value is assigned to *variable* if *variable* is of type character and zero  
 52 length, or is an array of size zero.

53 If *variable* is a pointer, the value of *expr* is assigned to the target of *variable*.

54 Both *variable* and *expr* may contain references to any portion of *variable*. For example, in the charac-  
 55 ter intrinsic assignment statement:

## Expressions and Assignment

1   STRING (2:5) = STRING (1:4)

2   the assignment of the first character of STRING to the second character does not affect the evalua-  
3   tion of STRING (1:4). For example, if the value of STRING prior to the assignment was 'ABCDEF',  
4   the value following the assignment is 'AABCDF'.

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

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

11   A = B

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

15   For example, the following program segment results in the values of the elements of array X being  
16   reversed:

17   REAL X (10)

18    ...

19   X (1:10) = X (10:1:-1)

20   A derived-type intrinsic assignment is performed as if each component of *expr* were assigned to  
21   the corresponding component of *variable* using pointer assignment (7.5.2) for pointer components,  
22   and intrinsic assignment for nonpointer components. The processor may perform the component-  
23   by-component assignment in any order or by any means that has the same effect.

24   For an example of a derived-type intrinsic assignment statement, if C and D are of the same  
25   derived type with a pointer component P and nonpointer components S, T, U, and V of type integer,  
26   logical, character, and another derived type, respectively, the intrinsic assignment

27   C = D

28   pointer assigns D % P to C % P and assigns D % S to C % S using the numeric intrinsic assignment  
29   statement, D % T to C % T using the logical intrinsic assignment statement, D % U to C % U using  
30   the character intrinsic assignment statement, and D % V to C % V using the derived-type intrinsic  
31   assignment statement.

32   When *variable* is a subobject, the assignment does not affect the definition status or value of other  
33   parts of the object. For example, if *variable* is an array section, the assignment does not affect the  
34   definition status or value of the elements of the parent array not specified by the array section.

35   **7.5.1.6 Interpretation of Defined Assignment Statements.** The interpretation of a defined  
36   assignment is provided by the subroutine that defines the operation.

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

38    (1) ~~The~~ subroutine is specified with a SUBROUTINE (12.5.2.3) or ENTRY (12.5.2.5) state-  
39    ~~ment~~ ment that specifies two dummy arguments,  $d_1$  and  $d_2$ ,

40    (2) An interface block (12.3.2.1) provides the subroutine with a *generic-spec* of  
41    ASSIGNMENT (=),

42    (3) The types of  $x_1$  and  $x_2$  are the same as those of the dummy arguments  $d_1$  and  $d_2$ , respec-  
43    tively,

44    (4) The type parameters, if any, of  $x_1$  and  $x_2$  match those of  $d_1$  and  $d_2$ , respectively, and

45    (5) The ranks of  $x_1$  and  $x_2$ , and their shapes if either or both are arrays, match those of  $d_1$   
46    and  $d_2$ , respectively.



1 Note that  $x_1$  and  $x_2$  must not have the same intrinsic type or both be numeric.

2 **7.5.2 Pointer Assignment.** Pointer assignment causes a pointer to become associated with a tar-  
3 get or causes its pointer association status to become disassociated or undefined.

4 R736 *pointer-assignment-stmt* is *pointer-object* => *target*

5 R737 *target* is *variable*  
6 or *expr*

7 Constraint: The *pointer-object* must have the POINTER attribute.

8 Constraint: The *variable* must have the TARGET attribute or be a subobject of an object with the  
9 TARGET attribute, or it must have the POINTER attribute ~~or be a subobject, other~~  
10 ~~than a structure component of an object with the POINTER attribute.~~

11 Constraint: The *target* must be of the same type, type parameters, and rank as the pointer.

12 Constraint: The *target* must not be an array section with a vector subscript.

13 Constraint: The *expr* must deliver a pointer result.

14 If the *target* is not a pointer, the pointer assignment statement associates the pointer with the tar-  
15 get. If the *target* is a pointer that is associated, the pointer is associated with the same object as the  
16 ~~target.~~ If the *target* is a pointer that is disassociated, the pointer also becomes disassociated. If the  
17 *target* is a pointer with undefined association status, the pointer also acquires an undefined associ-  
18 ation status.

19 Any previous association between the pointer and a <sup>object</sup> *target* is broken.

20 Pointer assignment for a pointer component of a structure also may take place by execution of a  
21 derived-type intrinsic assignment statement (7.5.1.5) or a defined assignment statement (7.5.1.5).

22 In addition to pointer assignment, a pointer becomes associated with a *target* by allocation of the  
23 pointer.

24 A pointer must not be referenced or defined unless it is associated with a *target* that may be refer-  
25 enced or defined.

26 The following are examples of pointer assignment statements.

```
27 NEW_NODE % LEFT => CURRENT_NODE
28 SIMPLE_NAME => STRUCTURE % SUBSTRUCT % COMPONENT
29 ROW => MAT2D (N, :)
30 WINDOW => MAT2D (I-1:I+1, J-1:J+1)
31 POINTER_OBJECT => POINTER_FUNCTION (ARG_1, ARG_2)
32 EVERY_OTHER => VECTOR (1:N:2)
```

33 **7.5.3 Masked Array Assignment—WHERE.** The masked array assignment is used to mask the  
34 evaluation of expressions and assignment of values in array assignment statements, according to  
35 the value of a logical array expression.

36 **7.5.3.1 General Form of the Masked Array Assignment.** A masked array assignment is either  
37 a WHERE statement or WHERE construct.

38 R738 *where-stmt* is WHERE (*mask-expr*) *assignment-stmt*

39 R739 *where-construct* is *where-construct-stmt*  
40 [ *assignment-stmt* ] ...  
41 [ *elsewhere-stmt*  
42 [ *assignment-stmt* ] ... ]  
43 *end-where-stmt*

44 R740 *where-construct-stmt* is WHERE (*mask-expr*)

45 R741 *mask-expr* is *logical-expr*

## Expressions and Assignment

1 R742 *elsewhere-stmt* is ELSEWHERE

2 R743 *end-where-stmt* is END WHERE

3 Constraint: In each *assignment-stmt*, the *mask-expr* and the variable being defined must be arrays  
4 of the same shape.

5 Constraint: The *assignment-stmt* must not be a defined assignment.

6 Examples of a masked array assignment are:

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

```
8 WHERE (PRESSURE <= 1.0)
```

```
9     PRESSURE = PRESSURE + INC_PRESSURE
```

```
10     TEMP = TEMP - 5.0
```

```
11 ELSEWHERE
```

```
12     RAINING = .TRUE.
```

```
13 END WHERE
```

14 **7.5.3.2 Interpretation of Masked Array Assignments.** When the *assignment-stmt* in a *where-stmt*  
15 is executed, the *expr* of the *assignment-stmt* is evaluated for all the elements where *mask-expr* is true  
16 and the result is assigned to the corresponding elements of *variable* according to the rules of intrinsic  
17 assignment (7.5.1). When a *where-construct* is executed, the *mask-expr* is evaluated and the result  
18 kept by the processor. Each *assignment-stmt* in the *where* block is evaluated, in sequence, as if it  
19 were WHERE (*mask-expr*) *assignment-stmt* and then each *assignment-stmt* in the ELSEWHERE block  
20 is evaluated, in sequence, as if it were WHERE (.NOT. *mask-expr*) *assignment-stmt*.

21 If a function reference appears in the *expr* of an assignment statement, the function must either be  
22 elemental (12.4.3) or have all scalar arguments.

23 If an elemental intrinsic operation or function reference occurs in the *expr* of an *assignment-stmt*, the  
24 operation is performed or the function is evaluated only for the elements corresponding to true  
25 values in *mask-expr* (false values after ELSEWHERE).

26 In a masked array assignment, only a WHERE statement or a WHERE construct statement may be  
27 a branch target statement. The value of *mask-expr* evaluated at the beginning of the masked array  
28 assignment governs the masking in the execution of the masked array assignment; subsequent  
29 changes to entities in *mask-expr* have no effect on the masking. The execution of a function refer-  
30 ence in the mask expression of a WHERE statement is permitted to affect entities in the assignment  
31 statement. Execution of an END WHERE has no effect.

32 Examples of function references in masked array assignments are:

```
33 WHERE (A > 0.0)
```

```
34     A = LOG (A)           ! LOG is invoked only for positive elements.
```

```
35     A = A / SUM (LOG (A)) ! LOG is invoked for all elements.
```

```
36 END WHERE
```

-----  
-----  
-----

## 8. EXECUTION CONTROL

- 1 The execution sequence may be controlled by constructs containing blocks and by certain execut-  
2 able statements that are used to alter the execution sequence.
- 3 **8.1 Executable Constructs Containing Blocks.** The following are executable constructs  
4 that contain blocks and may be used to control the execution sequence:
- 5 (1) IF Construct  
6 (2) CASE Construct  
7 (3) DO Construct
- 8 There is also a nonblock form of the DO construct.
- 9 A block is a sequence of executable constructs that is treated as a unit.
- 10 R801 *block* is [ *execution-part-construct* ] ...
- 11 Executable constructs may be used to control which blocks of a program are executed or how  
12 many times a block is executed. Blocks are always bounded by statements that are particular to  
13 the construct in which they are embedded; however, in some forms of the DO construct, a sequence of executable  
14 constructs without a terminating boundary statement must obey all other rules governing blocks (8.1.1). Note that a  
15 block need not contain any executable constructs. Execution of such a block has no effect.
- 16 Any of these three constructs may be named. If a construct is named, the name must be the first  
17 lexical token of the first statement of the construct and the last lexical token of the construct. In  
18 fixed source form, the name preceding the construct must be placed after column 6.
- 19 A statement belongs to the innermost construct in which it appears unless it contains a construct  
20 name, in which case it belongs to the named construct.
- 21 An example of a construct containing a block is:
- ```
22 IF (A > 0.0) THEN  
23     B = SQRT (A) ! These two statements  
24     C = LOG (A) ! form a block.  
25 END IF
```
- 26 **8.1.1 Rules Governing Blocks.**
- 27 **8.1.1.1 Executable Constructs in Blocks.** If a block contains an executable construct, the execut-  
28 able construct must be contained entirely within the block.
- 29 **8.1.1.2 Control Flow in Blocks.** Transfer of control to the interior of a block from outside the  
30 block is prohibited. Transfers within a block and transfers from the interior of a block to outside  
31 the block may occur. For example, if a statement inside the block has a statement label, a GO TO  
32 statement using that label may appear in the same block. Subroutine and function references may  
33 appear in a block (12.4.2, 12.4.3, 12.4.4, 12.4.5).
- 34 **8.1.1.3 Execution of a Block.** Execution of a block begins with the execution of the first execut-  
35 able construct in the block. Unless there is a transfer of control out of the block, the execution of  
36 the block is completed when the last executable construct in the sequence is executed. The action  
37 that takes place at the terminal boundary depends on the particular construct and on the block  
38 within that construct. It is usually a transfer of control.
- 39 **8.1.2 IF Construct.** The IF construct selects for execution no more than one of its constituent  
40 blocks. The IF statement controls the execution of a single statement (8.1.2.4).

## Execution Control

### 1 8.1.2.1 Form of the IF Construct.

|    |      |                     |    |                                                                          |
|----|------|---------------------|----|--------------------------------------------------------------------------|
| 2  | R802 | <i>if-construct</i> | is | <i>if-then-stmt</i>                                                      |
| 3  |      |                     |    | <i>block</i>                                                             |
| 4  |      |                     |    | [ <i>else-if-stmt</i>                                                    |
| 5  |      |                     |    | <i>block</i> ] ...                                                       |
| 6  |      |                     |    | [ <i>else-stmt</i>                                                       |
| 7  |      |                     |    | <i>block</i> ]                                                           |
| 8  |      |                     |    | <i>end-if-stmt</i>                                                       |
| 9  | R803 | <i>if-then-stmt</i> | is | [ <i>if-construct-name</i> : ] IF ( <i>scalar-logical-expr</i> ) THEN    |
| 10 | R804 | <i>else-if-stmt</i> | is | ELSE IF ( <i>scalar-logical-expr</i> ) THEN [ <i>if-construct-name</i> ] |
| 11 | R805 | <i>else-stmt</i>    | is | ELSE [ <i>if-construct-name</i> ]                                        |
| 12 | R806 | <i>end-if-stmt</i>  | is | END IF [ <i>if-construct-name</i> ]                                      |

13 Constraint: If the *if-then-stmt* of an *if-construct* is identified by an *if-construct-name*, the corre-  
14 sponding *end-if-stmt* must specify the same *if-construct-name*. If the *if-then-stmt* of an  
15 *if-construct* is not identified by an *if-construct-name*, the corresponding *end-if-stmt*  
16 must not specify an *if-construct-name*. If an *else-if-stmt* or *else-stmt* is identified by an  
17 *if-construct-name*, the corresponding *if-then-stmt* must specify the same *if-construct-*  
18 *name*.

19 8.1.2.2 Execution of an IF Construct. At most one of the blocks contained within the IF con-  
20 struct is executed. If there is an ELSE statement in the construct, exactly one of the blocks con-  
21 tained within the construct will be executed. The scalar logical expressions are evaluated in the  
22 order of their appearance in the construct until a true value is found or an ELSE statement or END  
23 IF statement is encountered. If a true value or an ELSE statement is found, the block immediately  
24 following is executed and this completes the execution of the construct. The scalar logical expres-  
25 sions in any remaining ELSE IF statements of the IF construct are not evaluated. If none of the  
26 evaluated expressions is true and there is no ELSE statement, the execution of the construct is com-  
27 pleted without the execution of any block within the construct.

28 An ELSE IF statement or an ELSE statement must not be a branch target statement. It is permissi-  
29 ble to branch to an END IF statement from within the IF construct, and also from outside the construct.  
30 Execution of an END IF statement has no effect.

### 31 8.1.2.3 Examples of IF Constructs.

```
32 IF (CVAR .EQ. 'RESET') THEN
33     I = 0; J = 0; K = 0
34 END IF
```

```
35 PROOF_DONE: IF (PROP) THEN
36     WRITE (3, '("QED")')
37     STOP
38 ELSE
39     PROP = NEXTPROP
40 END IF PROOF_DONE
```

```
41 IF (A .GT. 0) THEN
42     B = C/A
43     IF (B .GT. 0) THEN
44         D = 1.0
45     END IF
46 ELSE IF (C .GT. 0) THEN
47     B = A/C
48     D = -1.0
```

```

1 ELSE
2     B = ABS (MAX (A, C))
3     D = 0
4 END IF

```

5 **8.1.2.4 IF Statement.** The IF statement controls a single action statement (R216).

6 R807 *if-stmt* is IF (*scalar-logical-expr*) *action-stmt*

7 Constraint: The *action-stmt* in the *if-stmt* must not be an *if-stmt*, *end-program-stmt*, *end-function-*  
8 *stmt*, or *end-subroutine-stmt*.

9 Execution of an IF statement causes evaluation of the scalar logical expression. If the value of the  
10 expression is true, the action statement is executed. If the value is false, the action statement is not  
11 executed and execution continues as though a CONTINUE statement (8.3) were executed.

12 The execution of a function reference in the scalar logical expression is permitted to affect entities  
13 in the action statement.

14 An example of an IF statement is:

```

15 IF (A > 0.0) A = LOG (A)

```

16 **8.1.3 CASE Construct.** The CASE construct selects for execution at most one of its constituent  
17 blocks.

18 **8.1.3.1 Form of the CASE Construct.**

```

19 R808 case-construct is select-case-stmt
20 [ case-stmt
21 block ] ...
22 end-select-stmt

```

```

23 R809 select-case-stmt is [ case-construct-name : ] SELECT CASE ( case-expr )

```

```

24 R810 case-stmt is CASE case-selector [ case-construct-name ]

```

```

25 R811 end-select-stmt is END SELECT [ case-construct-name ]

```

26 Constraint: If the *select-case-stmt* of a *case-construct* is identified by a *case-construct-name*, the corre-  
27 sponding *end-select-stmt* must specify the same *case-construct-name*. If the *select-case-*  
28 *stmt* of a *case-construct* is not identified by a *case-construct-name*, the corresponding  
29 *end-select-stmt* must not specify a *case-construct-name*. If a *case-stmt* is identified by a  
30 *case-construct-name*, the corresponding *select-case-stmt* must specify the same *case-*  
31 *construct-name*.

```

32 R812 case-expr is scalar-int-expr
33 or scalar-char-expr
34 or scalar-logical-expr

```

```

35 R813 case-selector is ( case-value-range-list )
36 or DEFAULT

```

37 Constraint:— No more than one of the selectors of one of the CASE statements may be DEFAULT.

```

38 R814 case-value-range is case-value
39 or case-value :
40 or : case-value
41 or case-value : case-value

```

```

42 R815 case-value is scalar-int-initialization-expr
43 or scalar-char-initialization-expr
44 or scalar-logical-initialization-expr

```

45 Constraint: For a given *case-construct*, each *case-value* must be of the same type as *case-expr*. For  
46 character type, length differences are allowed, but the kind type parameters must be

## Execution Control

1 the same.

~~2~~ Constraint: A *case-value-range* using a colon must not be used if *case-expr* is of type logical.

3 Constraint: For a given *case-construct*, the *case-value-ranges* must not overlap; that is, there must  
4 be no possible value of the *case-expr* that matches more than one *case-value-range*.

5 **8.1.3.2 Execution of a CASE Construct.** The execution of the SELECT CASE statement causes  
6 the case expression to be evaluated. The resulting value is called the **case index**. For a case value  
7 range list, a match occurs if the case index matches any of the case value ranges in the list. For a  
8 case index with a value of *c*, a match is determined as follows:

9 (1) If the case value range contains a single value *v* without a colon, a match occurs for data  
10 type logical if the expression *c* .EQV. *v* is true, and a match occurs for data type integer  
11 or character if the expression *c* .EQ. *v* is true.

12 (2) If the case value range is of the form *low* : *high*, a match occurs if the expression *low* .LE.  
13 *c* .AND. *c* .LE. *high* is true.

14 (3) If the case value range is of the form *low* :, a match occurs if the expression *low* .LE. *c* is  
15 true.

16 (4) If the case value range is of the form : *high*, a match occurs if the expression *c* .LE. *high* is  
17 true.

18 (5) If no other selector matches and a DEFAULT selector is present, it matches the case  
19 index.

20 (6) If no other selector matches and the DEFAULT selector is absent, there is no match.  
21 Execution continues with the statement following the CASE construct.

22 The block following the CASE statement containing the matching selector, if any, is executed. This  
23 completes execution of the construct.

24 At most one of the blocks of a CASE construct is executed.

25 A CASE statement must not be a branch target statement. It is permissible to branch to an END  
26 SELECT statement only from within the CASE construct.

27 **8.1.3.3 Examples of CASE Constructs.** An integer signum function:

```
28 INTEGER FUNCTION SIGNUM (N)
29 SELECT CASE (N)
30 CASE (:-1)
31     SIGNUM = -1
32 CASE (0)
33     SIGNUM = 0
34 CASE (1:)
35     SIGNUM = 1
36 END SELECT
37 END
```

38 A code fragment to check for balanced parentheses:

```
39 CHARACTER (80) :: LINE
40     ...
41 LEVEL=0
42 DO I = 1, 80
43     CHECK_PARENS: SELECT CASE (LINE (I:I))
44     CASE ('(')
45         LEVEL = LEVEL + 1
```

```

1  CASE ('')
2      LEVEL = LEVEL - 1
3      IF (LEVEL .LT. 0) THEN
4          PRINT *, 'UNEXPECTED RIGHT PARENTHESIS'
5          EXIT
6      END IF
7  CASE DEFAULT
8      ! Ignore all other characters
9  END SELECT CHECK_PARENS
10 END DO
11 IF (LEVEL .GT. 0) THEN
12     PRINT *, 'MISSING RIGHT PARENTHESIS'
13 END IF

```

14 The following three fragments are equivalent:

```

15 IF (SILLY .EQ. 1) THEN
16     CALL THIS
17 ELSE
18     CALL THAT
19 END IF
20
21 SELECT CASE (SILLY .EQ. 1)
22 CASE (.TRUE.)
23     CALL THIS
24 CASE (.FALSE.)
25     CALL THAT
26 END SELECT
27
28 SELECT CASE (SILLY)
29 CASE DEFAULT
30     CALL THAT
31 CASE (1)
32     CALL THIS
33 END SELECT

```

34 A code fragment showing several selections of one block:

```

35 SELECT CASE (N)
36 CASE (1, 3:5, 8) ! Selects 1, 3, 4, 5, 8
37     CALL SUB
38 CASE DEFAULT
39     CALL OTHER
40 END SELECT

```

41 **8.1.4 DO Construct.** The DO construct specifies the repeated execution of a sequence of executable constructs. Such a repeated sequence is called a loop. The EXIT and CYCLE statements may be used to modify the execution of a loop.

44 The number of iterations of a loop may be determined at the beginning of execution of the DO construct, or may be left indefinite ("DO forever" or DO WHILE). In either case, an EXIT statement (8.1.4.4.4) anywhere in the DO construct may be executed to terminate the loop immediately. A particular iteration of the loop may be curtailed by executing a CYCLE statement (8.1.4.4.3).

48 **8.1.4.1 Forms of the DO Construct.** The DO construct may be written in either a block form or a nonblock form.

```

50 R816  do-construct           is block-do-construct
51                                     or nonblock-do-construct

```

## Execution Control

### 1 8.1.4.1.1 Form of the Block DO Construct.

|    |             |                                                                                                                                                                                                                        |                                                                           |
|----|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------|
| 2  | R817        | <i>block-do-construct</i>                                                                                                                                                                                              | is <i>do-stmt</i>                                                         |
| 3  |             |                                                                                                                                                                                                                        | <i>do-block</i>                                                           |
| 4  |             |                                                                                                                                                                                                                        | <i>end-do</i>                                                             |
| 5  | R818        | <i>do-stmt</i>                                                                                                                                                                                                         | is <i>label-do-stmt</i>                                                   |
| 6  |             |                                                                                                                                                                                                                        | or <i>nonlabel-do-stmt</i>                                                |
| 7  | R819        | <i>label-do-stmt</i>                                                                                                                                                                                                   | is [ <i>do-construct-name</i> : ] DO <i>label</i> [ <i>loop-control</i> ] |
| 8  | R820        | <i>nonlabel-do-stmt</i>                                                                                                                                                                                                | is [ <i>do-construct-name</i> : ] DO [ <i>loop-control</i> ]              |
| 9  | R821        | <i>loop-control</i>                                                                                                                                                                                                    | is [ , ] <i>do-variable</i> = <i>scalar-numeric-expr</i> , ■              |
| 10 |             |                                                                                                                                                                                                                        | ■ <i>scalar-numeric-expr</i> [ , <i>scalar-numeric-expr</i> ]             |
| 11 |             |                                                                                                                                                                                                                        | or [ , ] WHILE ( <i>scalar-logical-expr</i> )                             |
| 12 | R822        | <i>do-variable</i>                                                                                                                                                                                                     | is <i>scalar-variable</i>                                                 |
| 13 | Constraint: | The <i>do-variable</i> must be a named scalar variable of type integer, default real, or double precision real.                                                                                                        |                                                                           |
| 14 |             |                                                                                                                                                                                                                        |                                                                           |
| 15 | Constraint: | Each <i>scalar-numeric-expr</i> in <i>loop-control</i> must be of type integer, default real, or double precision real.                                                                                                |                                                                           |
| 16 |             |                                                                                                                                                                                                                        |                                                                           |
| 17 | R823        | <i>do-block</i>                                                                                                                                                                                                        | is <i>block</i>                                                           |
| 18 | R824        | <i>end-do</i>                                                                                                                                                                                                          | is <i>end-do-stmt</i>                                                     |
| 19 |             |                                                                                                                                                                                                                        | or <i>continue-stmt</i>                                                   |
| 20 | R825        | <i>end-do-stmt</i>                                                                                                                                                                                                     | is END DO [ <i>do-construct-name</i> ]                                    |
| 21 | Constraint: | If the <i>do-stmt</i> of a <i>block-do-construct</i> is identified by a <i>do-construct-name</i> , the corresponding <i>end-do</i> must be an <i>end-do-stmt</i> specifying the same <i>do-construct-name</i> . If the |                                                                           |
| 22 |             | <i>do-stmt</i> of a <i>block-do-construct</i> does not so specify a <i>do-construct-name</i> , the corresponding                                                                                                       |                                                                           |
| 23 |             | <i>end-do</i> must not specify a <i>do-construct-name</i> .                                                                                                                                                            |                                                                           |
| 24 |             |                                                                                                                                                                                                                        |                                                                           |
| 25 | Constraint: | If the <i>do-stmt</i> is a <i>nonlabel-do-stmt</i> , the corresponding <i>end-do</i> must be an <i>end-do-stmt</i> .                                                                                                   |                                                                           |
| 26 | Constraint: | If the <i>do-stmt</i> is a <i>label-do-stmt</i> , the corresponding <i>end-do</i> must be identified with the                                                                                                          |                                                                           |
| 27 |             | same <i>label</i> .                                                                                                                                                                                                    |                                                                           |

### 28 8.1.4.1.2 Form of the Nonblock DO Construct.

|    |             |                                                                                                                                                                                                                                                                                                                                            |                                            |
|----|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------|
| 29 | R826        | <i>nonblock-do-construct</i>                                                                                                                                                                                                                                                                                                               | is <i>action-term-do-construct</i>         |
| 30 |             |                                                                                                                                                                                                                                                                                                                                            | or <i>outer-shared-do-construct</i>        |
| 31 | R827        | <i>action-term-do-construct</i>                                                                                                                                                                                                                                                                                                            | is <i>label-do-stmt</i>                    |
| 32 |             |                                                                                                                                                                                                                                                                                                                                            | <i>do-body</i>                             |
| 33 |             |                                                                                                                                                                                                                                                                                                                                            | <i>do-term-action-stmt</i>                 |
| 34 | R828        | <i>do-body</i>                                                                                                                                                                                                                                                                                                                             | is [ <i>execution-part-construct</i> ] ... |
| 35 | R829        | <i>do-term-action-stmt</i>                                                                                                                                                                                                                                                                                                                 | is <i>action-stmt</i>                      |
| 36 | Constraint: | A <i>do-term-action-stmt</i> must not be a <i>continue-stmt</i> , a <i>goto-stmt</i> , a <i>return-stmt</i> , a <i>stop-stmt</i> , an <i>exit-stmt</i> , a <i>cycle-stmt</i> , an <i>end-function-stmt</i> , an <i>end-subroutine-stmt</i> , an <i>end-program-stmt</i> , an <i>arithmetic-if-stmt</i> , or an <i>assigned-goto-stmt</i> . |                                            |
| 37 |             |                                                                                                                                                                                                                                                                                                                                            |                                            |
| 38 |             |                                                                                                                                                                                                                                                                                                                                            |                                            |
| 39 | Constraint: | The <i>do-term-action-stmt</i> must be identified with a label and the corresponding <i>label-do-stmt</i> must refer to the                                                                                                                                                                                                                |                                            |
| 40 |             | same label.                                                                                                                                                                                                                                                                                                                                |                                            |
| 41 | R830        | <i>outer-shared-do-construct</i>                                                                                                                                                                                                                                                                                                           | is <i>label-do-stmt</i>                    |
| 42 |             |                                                                                                                                                                                                                                                                                                                                            | <i>do-body</i>                             |
| 43 |             |                                                                                                                                                                                                                                                                                                                                            | <i>shared-term-do-construct</i>            |
| 44 | R831        | <i>shared-term-do-construct</i>                                                                                                                                                                                                                                                                                                            | is <i>outer-shared-do-construct</i>        |
| 45 |             |                                                                                                                                                                                                                                                                                                                                            | or <i>inner-shared-do-construct</i>        |



1 R832 *inner-shared-do-construct* is *label-do-stmt*  
 2 *do-body*  
 3 *do-term-shared-stmt*

4 R833 *do-term-shared-stmt* is *action-stmt*

5 Constraint: A *do-term-shared-stmt* must not be a *goto-stmt*, a *return-stmt*, a *stop-stmt*, an *exit-stmt*, a *cycle-stmt*, an *end-*  
 6 *function-stmt*, an *end-subroutine-stmt*, an *end-program-stmt*, an *arithmetic-if-stmt*, or an *assigned-goto-stmt*.

7 Constraint: The *do-term-shared-stmt* must be identified with a label and all of the *label-do-stmts* of the *shared-term-do-*  
 8 *construct* must refer to the same label.

9 The *do-term-action-stmt*, *do-term-shared-stmt*, or *shared-term-do-construct* following the *do-body* of a nonblock DO construct is  
 10 called the DO termination of that construct.

11 Within a scoping unit, all DO constructs whose DO statements refer to the same label are nonblock DO constructs, and are  
 12 said to share the statement identified by that label.

13 **8.1.4.2 Range of the DO Construct.** The range of a block DO construct is the *do-block* which  
 14 must satisfy the rules for blocks (8.1.1). In particular, transfer of control to the interior of such a  
 15 block from outside the block is prohibited. It is permissible to branch to the *end-do* of a block DO  
 16 construct only from within the range of that DO construct.

17 The range of a nonblock DO construct consists of the *do-body* and the following DO termination. The end of such a range is  
 18 not bounded by a particular statement as for the other executable constructs (e.g., END IF); nevertheless, the range satisfies  
 19 the rules for blocks (8.1.1). Transfer of control into the *do-body* or to the DO termination from outside the range is prohib-  
 20 ited; in particular, it is permissible to branch to a *do-term-shared-stmt* only from within the range of the corresponding *inner-*  
 21 *shared-do-construct*.

22 **8.1.4.3 Active and Inactive DO Constructs.** A DO construct is either active or inactive. Ini-  
 23 tially inactive, a DO construct becomes active only when its DO statement is executed.

24 Once active, the DO construct becomes inactive only when the construct it specifies is terminated  
 25 (8.1.4.4.4). When an active DO construct becomes inactive, the *do-variable*, if any, retains its last  
 26 defined value.

27 **8.1.4.4 Execution of a DO Construct.** A DO construct specifies a loop, that is, a sequence of exe-  
 28 cutable constructs that is executed repeatedly. There are three phases in the execution of a DO  
 29 construct: initiation of the loop, execution of the loop range, and termination of the loop.

30 **8.1.4.4.1 Loop Initiation.** When the DO statement is executed, the DO construct becomes active.  
 31 If *loop-control* is

32  $[ , ] \textit{do-variable} = \textit{scalar-numeric-expr}_1, \textit{scalar-numeric-expr}_2 [ , \textit{scalar-numeric-expr}_3 ]$

33 the following steps are performed in sequence:

34 (1) The initial parameter  $m_1$ , the terminal parameter  $m_2$ , and the incrementation parameter  
 35  $m_3$  are established by evaluating *scalar-numeric-expr*<sub>1</sub>, *scalar-numeric-expr*<sub>2</sub>, and *scalar-*  
 36 *numeric-expr*<sub>3</sub>, respectively, including, if necessary, conversion to the type and kind type  
 37 parameter of the *do-variable* according to the rules for numeric conversion (Table 7.9). If  
 38 ~~*scalar-numeric-expr*<sub>3</sub>~~ does not appear,  $m_3$  is of type default integer and its value is 1. The  
 39 value  $m_3$  must not be zero.

40 (2) The DO variable becomes defined with the value of the initial parameter  $m_1$ .

41 (3) The iteration count is established and is the value of the expression

42 
$$\text{MAX}(\text{INT}((m_2 - m_1 + m_3) / m_3), 0)$$

43 Note that the iteration count is zero whenever:

44  $m_1 > m_2$  and  $m_3 > 0$ , or  
 45  $m_1 < m_2$  and  $m_3 < 0$ .

## Execution Control

1 If *loop-control* is omitted, no iteration count is calculated. The effect is as if a large positive iteration  
2 count, impossible to decrement to zero, were established. If *loop-control* is [ , ] WHILE (*scalar-*  
3 *logical-expr*), the effect is as if *loop-control* were omitted and the following statement inserted as the  
4 first statement of the *do-block*:

5 IF ( .NOT. (*scalar-logical-expr*) ) EXIT

6 At the completion of the execution of the DO statement, the execution cycle begins.

7 **8.1.4.4.2 The Execution Cycle.** The execution cycle of a DO construct consists of the following  
8 steps performed in sequence repeatedly until termination:

9 (1) The iteration count, if any, is tested. If the iteration count is zero, the loop terminates  
10 and the DO construct becomes inactive. If *loop-control* is [ , ] WHILE (*scalar-logical-expr*),  
11 the *scalar-logical-expr* is evaluated; if the value of this expression is false, the loop termi-  
12 nates and the DO construct becomes inactive. If, as a result, all of the DO constructs sharing the  
13 *do-term-shared-stmt* are inactive, the execution of all of these constructs is complete. However, if some of the  
14 DO constructs sharing the *do-term-shared-stmt* are active, execution continues with step (3) of the execution  
15 cycle of the active DO construct whose DO statement was most recently executed.

16 (2) If the iteration count is nonzero, the range of the loop is executed.

17 (3) The iteration count, if any, is decremented by one. The DO variable, if any, is incre-  
18 mented by the value of the incrementation parameter  $m_3$ .

19 Except for the incrementation of the DO variable that occurs in step (3), the DO variable must nei-  
20 ther be redefined nor become undefined while the DO construct is active.

21 **8.1.4.4.3 CYCLE Statement.** Step (2) in the above execution cycle may be curtailed by executing  
22 a CYCLE statement from within the range of the loop.

23 R834 *cycle-stmt* is CYCLE [ *do-construct-name* ]

24 Constraint: If a *cycle-stmt* refers to a *do-construct-name*, it must be within the range of that *do-*  
25 *construct*; otherwise, it must be within the range of at least one *do-construct*

26 A CYCLE statement belongs to a particular DO construct. If the CYCLE statement refers to a DO  
27 construct name, it belongs to that DO construct; otherwise, it belongs to the innermost DO con-  
28 struct in which it appears.

29 Execution of a CYCLE statement causes immediate progression to step (3) of the current execution  
30 cycle of the DO construct to which it belongs. If this construct is a nonblock DO construct, the *do-term-action-stmt*  
31 or *do-term-shared-stmt* is not executed.

32 In a block DO construct, a transfer of control to the *end-do* has the same effect as execution of a  
33 CYCLE statement belonging to that construct. In a nonblock DO construct, transfer of control to the *do-term-*  
34 *action-stmt* or *do-term-shared-stmt* causes that statement or construct itself to be executed. Unless a further transfer of control  
35 results, step (3) of the current execution cycle of the DO construct is then executed.

36 **8.1.4.4.4 Loop Termination.** The EXIT statement provides one way of terminating a loop.

37 R835 *exit-stmt* is EXIT [ *do-construct-name* ]

38 Constraint: If an *exit-stmt* refers to a *do-construct-name*, it must be within the range of that *do-*  
39 *construct*; otherwise, it must be within the range of at least one *do-construct*.

40 An EXIT statement belongs to a particular DO construct. If the EXIT statement refers to a DO con-  
41 struct name, it belongs to that DO construct; otherwise, it belongs to the innermost DO construct in  
42 which it appears.

43 The loop terminates, and the DO construct becomes inactive, when any of the following occurs:

44 (1) Determination that the iteration count is zero when tested during step (1) of the above  
45 execution cycle

- 1       (2) Execution of an EXIT statement belonging to the DO construct  
 2       (3) Execution of an EXIT statement or a CYCLE statement that is within the range of the  
 3       DO construct, but that belongs to an outer DO construct  
 4       (4) Transfer of control from a statement within the range of a DO construct to a statement  
 5       that is neither the *end-do* nor within the range of the same DO construct  
 6       (5) Execution of a RETURN statement within the range of the DO construct  
 7       (6) Execution of a STOP statement anywhere in the program; or termination of the pro-  
 8       gram for any other reason.
- 9       When a DO construct becomes inactive, the DO-variable, if any, of the DO construct retains its last  
 10      defined value.

11   **8.1.4.5 Examples of DO Constructs.** The following are all valid examples of block DO con-  
 12   structs.

13   Example 1:

```
14 DO I = 1, M
15   DO J = 1, N
16     C (I, J) = SUM (A (I, J, :) * B (:, I, J))
17   END DO
18 END DO
```

19   The above program fragment computes a tensor product of two arrays.

20   Example 2:

```
21 READ (IUN, '(1X, G14.7)', IOSTAT = IOS) X
22 DO WHILE (IOS .EQ. 0)
23   IF (X .GE. 0.) THEN
24     CALL SUBA (X)
25     CALL SUBB (X)
26     ...
27     CALL SUBZ (X)
28   ENDIF
29   READ (IUN, '(1X, G14.7)', IOSTAT = IOS) X
30 END DO
```

31   The above program fragment contains a DO construct that uses the WHILE form of *loop-control*.  
 32   The loop will continue to execute until an end-of-file or input/output error is encountered, at  
 33   which point the DO statement terminates the loop. When a negative value of X is read, the pro-  
 34   gram skips immediately to the next READ statement, bypassing most of the range of the loop.

35   Example 3:

```
36 DO ! A "DO WHILE + 1/2" loop
37   READ (IUN, '(1X, G14.7)', IOSTAT = IOS) X
38   IF (IOS .NE. 0) EXIT
39   IF (X < 0.) CYCLE
40   CALL SUBA (X)
41   CALL SUBB (X)
42   ...
43   CALL SUBZ (X)
44 END DO
```

45   Example 3 behaves exactly the same as example 2. However, the READ statement has been moved  
 46   to the interior of the range, so that only one READ statement is required. Also, a CYCLE statement  
 47   has been used to avoid an extra level of IF nesting.

48   Example 4:

## Execution Control

```
1      SUM = 0.0
2      READ (IUN) N
3      OUTER: DO L = 1, N          ! A DO with a construct name
4          READ (IUN) IQUAL, M, ARRAY (1:M)
5          IF (IQUAL < IQUAL_MIN) CYCLE OUTER    ! Skip inner loop
6          INNER: DO 40 I = 1, M      ! A DO with a label and a name
7              CALL CALCULATE (ARRAY (I), RESULT)
8              IF (RESULT < 0.0) CYCLE
9              SUM = SUM + RESULT
10             IF (SUM > SUM_MAX) EXIT OUTER
11     40  END DO INNER
12     END DO OUTER
```

13 The outer loop has an iteration count of MAX (N, 0), and will execute that number of times unless  
14 SUM exceeds SUM\_MAX, in which case the EXIT OUTER statement terminates both loops. The  
15 inner loop is skipped by the first CYCLE statement if the quality flag, IQUAL, is too low. If CAL-  
16 CULATE returns a negative RESULT, the second CYCLE statement prevents it from being  
17 summed. Note that both loops have construct names and the inner loop also has a label. A con-  
18 struct name is required on the EXIT statement in order to terminate both loops, but is optional on  
19 the CYCLE statements because each belongs to its innermost loop.

### 20 Example 5:

```
21      N = 0
22      DO 50, I = 1, 10
23          J = I
24          DO K = 1, 5
25              L = K
26              N = N + 1    ! This statement executes 50 times
27          END DO          ! Nonlabeled DO inside a labeled DO
28     50 CONTINUE
```

29 After execution of the above program fragment, I = 11, J = 10, K = 6, L = 5, and N = 50.

### 30 Example 6:

```
31      N = 0
32      DO I = 1, 10
33          J = I
34          DO 60, K = 5, 1    ! This inner loop is never executed
35              L = K
36              N = N + 1
37     60  CONTINUE          ! Labeled DO inside a nonlabeled DO
38     END DO
```

39 After execution of the above program fragment, I = 11, J = 10, K = 5, N = 0, and L is not defined by  
40 these statements.

41 The following are all valid examples of nonblock DO constructs:

### 42 Example 7:

```
43      DO 70
44          READ (IUN, '(1X, G14.7)', IOSTAT = IOS) X
45          IF (IOS .NE. 0) EXIT
46          IF (X < 0.) GOTO 70
47          CALL SUBA (X)
48          CALL SUBB (X)
49          ...
50          CALL SUBY (X)
51          CYCLE
52     70  CALL SUBNEG (X)    ! SUBNEG called only when X < 0.
```

1 This is not a block DO construct because it ends with a statement other than END DO or CONTINUE. The loop will continue to execute until an end-of-file condition or input/output error occurs.

3 Example 8:

```

4     SUM = 0.0
5     READ (IUN) N
6     DO 80, L = 1, N
7         READ (IUN) IQUAL, M, ARRAY (1:M)
8         IF (IQUAL < IQUAL_MIN) M = 0 ! Skip inner loop
9         DO 80 I = 1, M
10            CALL CALCULATE (ARRAY (I), RESULT)
11            IF (RESULT < 0.) CYCLE
12            SUM = SUM + RESULT
13            IF (SUM > SUM_MAX) GOTO 81
14     80 CONTINUE ! This CONTINUE is shared by both loops
15     81 CONTINUE

```

16 This example is similar to Example 3 above, except that the two loops are not block DO constructs because they share the CONTINUE statement with the label 80. The terminal construct of the outer DO is the entire inner DO construct. The inner loop is skipped by forcing M to zero. If SUM grows too large, both loops are terminated by branching to the CONTINUE statement labeled 81. The CYCLE statement in the inner loop is used to skip negative values of RESULT.

20 Example 9:

```

21     N = 0
22     DO 100 I = 1, 10
23         J = I
24         DO 100 K = 1, 5
25             L = K
26     100     N = N + 1 ! This statement executes 50 times

```

27 In this example, the two loops share an assignment statement. After execution of this program fragment, I = 11, J = 10, K = 6, L = 5, and N = 50.

29 Example 10:

```

30     N = 0
31     DO 200 I = 1, 10
32         J = I
33         DO 200 K = 5, 1 ! This inner loop is never executed
34             L = K
35     200     N = N + 1

```

36 This example is very similar to the previous one, except that the inner loop is never executed. After execution of this program fragment, I = 11, J = 10, K = 5, N = 0, and L is not defined by these statements.

38 **8.2 Branching.** Branching is used to alter the normal execution sequence. A branch causes a transfer of control from one statement in a scoping unit to a labeled branch target statement in the same scoping unit. A branch target statement is an *action-stmt*, an *if-then-stmt*, an *end-if-stmt*, a *select-case-stmt*, an *end-select-stmt*, a *do-stmt*, an *end-do-stmt*, a *do-term-action-stmt*, a *do-term-shared-stmt*, or a *where-construct-stmt*.

43 It is permissible to branch to an END SELECT statement only from within its CASE construct.

44 It is permissible to branch to an END IF statement from within its IF construct, and also from outside the construct.

46 It is permissible to branch to an *end-do-stmt*, a *do-term-action-stmt*, or a *do-term-shared-stmt* only from within its DO construct.

## Execution Control

1 **8.2.1 Statement Labels.** A statement label provides a means of referring to an individual state-  
2 ment. Any statement not forming part of another statement may be identified with a label, but  
3 only branch target statements, FORMAT statements, and DO terminations may be referred to by  
4 the use of statement labels (3.2.5).

### 5 8.2.2 GO TO Statement.

6 R836 *goto-stmt* is GO TO *label*

7 Constraint: The *label* must be the statement label of a branch target statement that appears in the  
8 same scoping unit as the *goto-stmt*.

9 Execution of a GO TO statement causes a transfer of control so that the branch target statement  
10 identified by the label is executed next.

### 11 8.2.3 Computed GO TO Statement.

12 R837 *computed-goto-stmt* is GO TO (*label-list*) [, ] *scalar-int-expr*

13 Constraint: Each *label* in *label-list* must be the statement label of a branch target statement that  
14 appears in the same scoping unit as the *computed-goto-stmt*.

15 The same statement label may appear more than once in a label list.

16 Execution of a computed GO TO statement causes evaluation of the scalar integer expression. If  
17 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  
18 occurs so that the next statement executed is the one identified by the  $i$ th label in the list of labels.  
19 If  $i$  is less than 1 or greater than  $n$ , the execution sequence continues as though a CONTINUE state-  
20 ment were executed.

### 21 8.2.4 ASSIGN and Assigned GO TO Statement.

22 R838 *assign-stmt* is ASSIGN *label* TO *scalar-int-variable*

23 Constraint: The *label* must be the statement label of a branch target statement or *format-stmt* that appears in the same  
24 scoping unit as the *assign-stmt*.

25 Constraint: *scalar-int-variable* must be of type default integer, *and must not be an array element*

26 R839 *assigned-goto-stmt* is GO TO *scalar-int-variable* [, ] (*label-list*) ]

27 Constraint: Each *label* in *label-list* must be the statement label of a branch target statement that appears in the same  
28 scoping unit as the *assigned-goto-stmt*.

29 Constraint: *scalar-int-variable* must be of type default integer, *and must not be an array element*

30 Execution of an ASSIGN statement causes a statement label to be assigned to an integer variable. While defined with a  
31 statement label value, the integer variable may be referenced only in the context of an assigned GO TO statement or as a for-  
32 mat specifier in an input/output statement. An integer variable defined with a statement label value may be redefined with  
33 a statement label value or an integer value.

34 When an input/output statement containing the integer variable as a format specifier (9.4.1.1) is executed, the integer vari-  
35 able must be defined with the label of a FORMAT statement.

36 At the time of execution of an assigned GO TO statement, the integer variable must be defined with the value of a statement  
37 label of a branch target statement that appears in the same scoping unit. Note that the variable may be defined with a state-  
38 ment label value only by an ASSIGN statement in the same scoping unit as the assigned GO TO statement.

39 The execution of the assigned GO TO statement causes a transfer of control so that the branch target statement identified by  
40 the statement label currently assigned to the integer variable is executed next.

41 If the parenthesized list is present, the statement label assigned to the integer variable must be one of the statement labels in  
42 the list. A label may appear more than once in the label list of an assigned GOTO statement.

Conflict  
with F77  
or OK?

1 **8.2.5 Arithmetic IF Statement.**2 R840 *arithmetic-if-stmt* is IF ( *scalar-numeric-expr* ) *label* , *label* , *label*3 Constraint: Each *label* must be the label of a branch target statement that appears in the same scoping unit as the  
4 *arithmetic-if-stmt*.5 Constraint: The *scalar-numeric-expr* must not be of type complex.

6 The same label may appear more than once in one arithmetic IF statement.

7 Execution of an arithmetic IF statement causes evaluation of the numeric expression followed by a transfer of control. The  
8 branch target statement identified by the first label, the second label, or the third label is executed next as the value of the  
9 numeric expression is less than zero, equal to zero, or greater than zero, respectively.10 **8.3 CONTINUE Statement.**

11 Execution of a CONTINUE statement has no effect.

12 R841 *continue-stmt* is CONTINUE13 **8.4 STOP Statement.**14 R842 *stop-stmt* is STOP [ *stop-code* ]15 R843 *stop-code* is *scalar-char-constant*  
16 or *digit* [ *digit* [ *digit* [ *digit* [ *digit* ] ] ] ]17 Constraint: *scalar-char-constant* must be of type default character.18 Execution of a STOP statement causes termination of execution of the executable program. At the  
19 time of termination, the stop code, if any, is available in a processor-dependent manner. Leading  
20 zero digits are not significant.21 **8.5 PAUSE Statement.**22 R844 *pause-stmt* is PAUSE [ *stop-code* ]23 Execution of a PAUSE statement causes a suspension of execution of the executable program. Execution must be resum-  
24 able. At the time of suspension of execution, the stop code, if any, is available in a processor-dependent manner. Leading  
25 zero digits in the stop code are not significant. Resumption of execution is not under control of the program. If execution is  
26 resumed, the execution sequence continues as though a CONTINUE statement were executed.

-----  
-----  
-----  
-----



## 9. INPUT/OUTPUT STATEMENTS

1 Input statements provide the means of transferring data from external media to internal storage or  
2 from an internal file to internal storage. This process is called reading. Output statements pro-  
3 vide the means of transferring data from internal storage to external media or from internal stor-  
4 age to an internal file. This process is called writing. Some input/output statements specify that  
5 editing of the data is to be performed.

6 In addition to the statements that transfer data, there are auxiliary input/output statements to  
7 manipulate the external medium, or to describe or inquire about the properties of the connection  
8 to the external medium.

9 The input/output statements are the OPEN, CLOSE, READ, WRITE, PRINT, BACKSPACE,  
10 ENDFILE, REWIND, and INQUIRE statements.

11 The READ statement is a **data transfer input statement**. The WRITE statement and the PRINT  
12 statement are **data transfer output statements**. The OPEN statement and the CLOSE statement are  
13 **file connection statements**. The INQUIRE statement is a **file inquiry statement**. The BACK-  
14 SPACE, ENDFILE, and REWIND statements are **file positioning statements**.

15 **9.1 Records.** A record is a sequence of values or a sequence of characters. For example, a line  
16 on a terminal is usually considered to be a record. However, a record does not necessarily corre-  
17 spond to a physical entity. There are three kinds of records:

- 18 (1) Formatted
- 19 (2) Unformatted
- 20 (3) Endfile

21 **9.1.1 Formatted Record.** A formatted record consists of a sequence of characters that are capa-  
22 ble of representation in the processor; however, a processor may prohibit some control characters  
23 (3.1) from appearing in a formatted record. The length of a formatted record is measured in char-  
24 acters and depends primarily on the number of characters put into the record when it is written.  
25 However, it may depend on the processor and the external medium. The length may be zero. For-  
26 matted records may be read or written only by formatted input/output statements.

27 Formatted records may be prepared by means other than Fortran; for example, by some manual  
28 input device.

29 **9.1.2 Unformatted Record.** An unformatted record consists of a sequence of values in a  
30 processor-dependent form and may contain data of any type or may contain no data. The length  
31 of an unformatted record is measured in processor-dependent units and depends on the output list  
32 (9.4.2) used when it is written, as well as on the processor and the external medium. The length  
33 may be zero. Unformatted records may be read or written only by unformatted input/output  
34 statements.

35 **9.1.3 Endfile Record.** An endfile record is written explicitly by the ENDFILE statement; the file  
36 must be connected for sequential access. An endfile record is written implicitly to a file connected  
37 for sequential access when the last reference to the file is a data transfer output statement or a file  
38 positioning statement, and:

- 39 (1) A REWIND or BACKSPACE statement references the unit to which the file is con-  
40 nected, or
- 41 (2) The unit (file) is closed, either explicitly by a CLOSE statement or implicitly by a pro-  
42 gram termination not caused by an error condition.

43 An endfile record may occur only as the last record of a file. An endfile record does not have a  
44 length property.

## Input/Output Statements

1   **9.2 Files.** A file is a sequence of records.

2   There are two kinds of files:

3       (1) External

4       (2) Internal

5   **9.2.1 External Files.** An external file is any file that exists in a medium external to the executable  
6 program.

7   At any given time, there is a processor-dependent set of allowed access methods, a processor-  
8 dependent set of allowed forms, a processor-dependent set of allowed actions, and a processor-  
9 dependent set of allowed record lengths for a file.

10 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  
11 character string. The set of allowable names for a file is processor dependent.

12 An external file that is connected to a unit has a **position property** (9.2.1.3).

13 **9.2.1.1 File Existence.** At any given time, there is a processor-dependent set of external files that  
14 are said to exist for an executable program. A file may be known to the processor, yet not exist for  
15 an executable program at a particular time. For example, there may be security reasons that pre-  
16 vent a file from existing for an executable program. A file may exist and contain no records; an  
17 example is a newly created file not yet written.

18 To create a file means to cause a file to exist that did not exist previously. To delete a file means  
19 to terminate the existence of the file.

20 All input/output statements may refer to files that exist. An INQUIRE, OPEN, CLOSE, WRITE,  
21 PRINT, REWIND, or ENDFILE statement also may refer to a file that does not exist. Execution of a  
22 WRITE or PRINT statement referring to a preconnected file that does not exist creates the file.

23 **9.2.1.2 File Access.** There are two methods of accessing the records of an external file, sequential  
24 and direct. Some files may have more than one allowed access method; other files may be  
25 restricted to one access method. For example, a processor may allow only sequential access to a  
26 file on magnetic tape. Thus, the set of allowed access methods depends on the file and the proces-  
27 sor.

28 The method of accessing the file is determined when the file is connected to a unit (9.3.2) or when  
29 the file is created if the file is preconnected (9.3.3).

30 **9.2.1.2.1 Sequential Access.** When connected for sequential access, an external file has the fol-  
31 lowing properties:

32       (1) The order of the records is the order in which they were written if the direct access  
33 method is not a member of the set of allowed access methods for the file. If the direct  
34 access method is also a member of the set of allowed access methods for the file, the  
35 order of the records is the same as that specified for direct access. In this case, the first  
36 record accessed by sequential access is the record whose record number is 1 for direct  
37 access. The second record accessed by sequential access is the record whose record  
38 number is 2 for direct access, etc. A record that has not been written since the file was  
39 created must not be read.

40       (2) The records of the file are either all formatted or all unformatted, except that the last  
41 record of the file may be an endfile record. Unless the previous reference to the file was  
42 a data transfer output statement or a file positioning statement, the last record, if any, of  
43 the file must be an endfile record.

44       (3) The records of the file must not be read or written by direct access input/output state-  
45 ments.

- 1 **9.2.1.2.2 Direct Access.** When connected for direct access, an external file has the following  
 2 properties:
- 3 (1) Each record of the file is uniquely identified by a positive integer called the **record num-**  
 4 **ber.** The record number of a record is specified when the record is written. Once estab-  
 5 lished, the record number of a record can never be changed. Note that a record may not  
 6 be deleted; however, a record may be rewritten. The order of the records is the order of  
 7 their record numbers.
  - 8 (2) The records of the file are either all formatted or all unformatted. If the sequential  
 9 access method is also a member of the set of allowed access methods for the file, its  
 10 endfile record, if any, is not considered to be part of the file while it is connected for  
 11 direct access. If the sequential access method is not a member of the set of allowed  
 12 access methods for the file, the file must not contain an endfile record.
  - 13 (3) Reading and writing records is accomplished only by direct access input/output state-  
 14 ments.
  - 15 (4) All records of the file have the same length.
  - 16 (5) Records need not be read or written in the order of their record numbers. Any record  
 17 may be written into the file while it is connected to a unit. For example, it is permissible  
 18 to write record 3, even though records 1 and 2 have not been written. Any record may  
 19 be read from the file while it is connected to a unit, provided that the record has been  
 20 written since the file was created.
  - 21 (6) The records of the file must not be read or written using list-directed formatting (10.8),  
 22 namelist formatting (10.9), or a nonadvancing input/output statement.
- 23 **9.2.1.3 File Position.** Execution of certain input/output statements affects the position of an  
 24 external file. Certain circumstances can cause the position of a file to become indeterminate.
- 25 The **initial point** of a file is the position just before the first record. The **terminal point** is the posi-  
 26 tion just after the last record. If there are no records in the file, the initial point and the terminal  
 27 point are the same position.
- 28 If a file is positioned within a record, that record is the **current record**; otherwise, there is no cur-  
 29 rent record.
- 30 Let  $n$  be the number of records in the file. If  $1 < i \leq n$  and a file is positioned within the  $i$ th record  
 31 or between the  $(i - 1)$ th record and the  $i$ th record, the  $(i - 1)$ th record is the **preceding record**. If  $n \geq$   
 32 1 and the file is positioned at its terminal point, the preceding record is the  $n$ th and last record. If  $n$   
 33 = 0 or if a file is positioned at its initial point or within the first record, there is no preceding  
 34 record.
- 35 If  $1 \leq i < n$  and a file is positioned within the  $i$ th record or between the  $i$ th and  $(i + 1)$ th record, the  
 36  $(i + 1)$ th record is the **next record**. If  $n \geq 1$  and the file is positioned at its initial point, the first  
 37 record is the next record. If  $n = 0$  or if a file is positioned at its terminal point or within the  $n$ th  
 38 (last) record, there is no next record.
- 39 **9.2.1.3.1 Advancing and Nonadvancing Input/Output.** An advancing input/output state-  
 40 ment **always** positions the file after the last record read or written, unless there is an error condi-  
 41 tion.
- 42 A **nonadvancing input/output statement** may position the file at a character position within the  
 43 current record. Using nonadvancing input/output, it is possible to read or write a record of the  
 44 file by a sequence of input/output statements, each accessing a portion of the record. It is also  
 45 possible to read variable-length records and be notified of their lengths.
- 46 **9.2.1.3.2 File Position Prior to Data Transfer.** The positioning of the file prior to data transfer  
 47 depends on the method of access: sequential or direct.

## Input/Output Statements

- 1 For sequential access on input, if there is a current record, the file position is not changed. Other-
- 2 wise, the file is positioned at the beginning of the next record and this record becomes the current
- 3 record. Input must not occur if there is no next record or if there is a current record and the last
- 4 data transfer statement accessing the file performed output.
  
- 5 If the file contains an endfile record, the file must not be positioned after the endfile record prior to
- 6 data transfer. However, a REWIND or BACKSPACE statement may be used to reposition the file.
  
- 7 For sequential access on output, if there is a current record, the file position is not changed and the
- 8 current record becomes the last record of the file. Otherwise, a new record is created as the next
- 9 record of the file; this new record becomes the last and current record of the file and the file is posi-
- 10 tioned at the beginning of this record.
  
- 11 For direct access, the file is positioned at the beginning of the record specified by the record speci-
- 12 fier. This record becomes the current record.

13 **9.2.1.3.3 File Position after Data Transfer.** If an error condition (9.4.3) occurred, the position of  
14 the file is indeterminate. If no error condition occurred, but an end-of-file condition (9.4.3)  
15 occurred as a result of reading an endfile record, the file is positioned after the endfile record.

16 For nonadvancing input, if no error condition or end-of-file condition occurred, but an end-of-  
17 record condition (9.4.3) occurred, the file is positioned after the record just read. If no error condi-  
18 tion, end-of-file condition, or end-of-record condition occurred in a nonadvancing input statement,  
19 the file position is not changed. If no error condition occurred in a nonadvancing output state-  
20 ment, the file position is not changed. In all other cases, the file is positioned after the record just  
21 read or written and that record becomes the preceding record.

22 **9.2.2 Internal Files.** Internal files provide a means of transferring and converting data from inter-  
23 nal storage to internal storage.

24 **9.2.2.1 Internal File Properties.** An internal file has the following properties:

- 25 (1) The file is a variable of default character type that is not an array section with a vector  
26 subscript.
- 27 (2) A record of an internal file is a scalar character variable.
- 28 (3) If the file is a scalar character variable, it consists of a single record whose length is the  
29 same as the length of the scalar character variable. If the file is a character array, it is  
30 treated as a sequence of character array elements. Each array element, if any, is a record  
31 of the file. The ordering of the records of the file is the same as the ordering of the array  
32 elements in the array (6.2.2.2) or the array section (6.2.2.3). Every record of the file has  
33 the same length, which is the length of an array element in the array.
- 34 (4) A record of the internal file becomes defined by writing the record. If the number of  
35 characters written in a record is less than the length of the record, the remaining portion  
36 of the record is filled with blanks. The number of characters to be written must not  
37 exceed the length of the record.
- 38 (5) ~~A record~~ may be read only if the record is defined.
- 39 (6) ~~A record~~ of an internal file may become defined (or undefined) by means other than an  
40 output statement. For example, the character variable may become defined by a charac-  
41 ter assignment statement.
- 42 (7) An internal file is always positioned at the beginning of the first record prior to data  
43 transfer. This record becomes the current record.
- 44 (8) On input, blanks are treated as though the format had an initial BN edit descriptor  
45 (10.6.6) and records are padded with blanks if necessary (9.4.4.4.2).
- 46 (9) On list-directed output, character constants are not delimited (10.8.2).

1 **9.2.2.2 Internal File Restrictions.** An internal file has the following restrictions:

- 2 (1) Reading and writing records must be accomplished only by sequential access formatted  
3 input/output statements that do not specify namelist formatting.
- 4 (2) An internal file must not be specified in a file connection statement, a file positioning  
5 statement, or a file inquiry statement.

6 **9.3 File Connection.** A unit, specified by an *io-unit*, provides a means for referring to a file.

7 R901 *io-unit* is *external-file-unit*

8 or \*

9 or *internal-file-unit*

10 R902 *external-file-unit* is *scalar-int-expr*

11 R903 *internal-file-unit* is *default-char-variable*

12 Constraint: The *char-variable* must not be an array section with a vector subscript.

13 A unit is either an external unit or an internal unit. An external unit is used to refer to an external  
14 file and is specified by an *external-file-unit* or an asterisk. An internal unit is used to refer to an  
15 internal file and is specified by an *internal-file-unit*.

16 If a character variable that identifies an internal file unit is a pointer, it must be associated. If the  
17 character variable is an allocatable array or a subobject of such an array, the array must be cur-  
18 rently allocated.

19 A scalar integer expression that identifies an external file unit must be zero or positive.

20 The *io-unit* in a file positioning statement, a file connection statement, or a file inquiry statement  
21 must be an *external-file-unit*.

22 The external unit identified by the value of the *scalar-int-expr* is the same external unit in all pro-  
23 gram units of the executable program. In the example:

```
24 SUBROUTINE A
25     READ (6) X
26     .
27     ..
28     .
29 SUBROUTINE B
30     N = 6
31     REWIND N
```

32 the value 6 used in both program units identifies the same external unit.

33 An asterisk identifies particular processor-dependent external units that are preconnected for for-  
34 matted sequential access (9.4.4.2).

35 **9.3.1 Unit Existence.** At any given time, there is a processor-dependent set of external units that  
36 are said to exist for an executable program.

37 All input/output statements may refer to units that exist. The INQUIRE statement and the CLOSE  
38 statement also may refer to units that do not exist.

39 **9.3.2 Connection of a File to a Unit.** An external unit has a property of being connected or not  
40 connected. If connected, it refers to an external file. An external unit may become connected by  
41 preconnection or by the execution of an OPEN statement. The property of connection is symmet-  
42 ric; if a unit is connected to a file, the file is connected to the unit.

43 All input/output statements except an OPEN, a CLOSE, or an INQUIRE statement must refer to a  
44 unit that is connected to a file and thereby make use of or affect that file.

## Input/Output Statements

- 1 A file may be connected and not exist. An example is a preconnected external file that has not yet  
2 been written (9.2.1.1).
- 3 A unit must not be connected to more than one file at the same time, and a file must not be con-  
4 nected to more than one unit at the same time. However, means are provided to change the status  
5 of an external unit and to connect a unit to a different file.
- 6 After an external unit has been disconnected by the execution of a CLOSE statement, it may be  
7 connected again within the same executable program to the same file or to a different file. After an  
8 external file has been disconnected by the execution of a CLOSE statement, it may be connected  
9 again within the same executable program to the same unit or to a different unit. Note, however,  
10 that the only means of referencing a file that has been disconnected is by the appearance of its  
11 name in an OPEN or INQUIRE statement. There may be no means of reconnecting an unnamed  
12 file once it is disconnected.
- 13 An internal unit is always connected to the internal file designated by the variable of default char-  
14 acter type that identifies the unit.

15 **9.3.3 Preconnection.** Preconnection means that the unit is connected to a file at the beginning of  
16 execution of the executable program and therefore it may be specified in input/output statements  
17 without the prior execution of an OPEN statement.

18 **9.3.4 The OPEN Statement.** An OPEN statement initiates or modifies the connection between an  
19 external file and a specified unit. The OPEN statement may be used to connect an existing file to a  
20 unit, create a file that is preconnected, create a file and connect it to a unit, or change certain speci-  
21 fiers of a connection between a file and a unit.

22 An external unit may be connected by an OPEN statement in any program unit of an executable  
23 program and, once connected, a reference to it may appear in any program unit of the executable  
24 program.

25 If a unit is connected to a file that exists, execution of an OPEN statement for that unit is permitted.  
26 If the FILE= specifier is not included in such an OPEN statement, the file to be connected to the  
27 unit is the same as the file to which the unit is already connected.

28 If the file to be connected to the unit does not exist but is the same as the file to which the unit is  
29 preconnected, the properties specified by an OPEN statement become a part of the connection.

30 If the file to be connected to the unit is not the same as the file to which the unit is connected, the  
31 effect is as if a CLOSE statement without a STATUS= specifier had been executed for the unit  
32 immediately prior to the execution of an OPEN statement.

33 If the file to be connected to the unit is the same as the file to which the unit is connected, only the  
34 BLANK=, DELIM=, PAD=, ERR=, and IOSTAT= specifiers may have values different from those  
35 currently in effect. Execution of such an OPEN statement causes any new value of the BLANK=,  
36 DELIM=, or PAD= specifiers to be in effect, but does not cause any change in any of the unspeci-  
37 fied specifiers and the position of the file is unaffected. The ERR= and IOSTAT= specifiers from  
38 any previously executed OPEN statement have no effect on any currently executed OPEN state-  
39 ment.

40 If a file is already connected to a unit, execution of an OPEN statement on that file and a different  
41 unit is not permitted.

42 R904 *open-stmt* is OPEN ( *connect-spec-list* )

43 R905 *connect-spec* is [ UNIT= ] *external-file-unit*  
44 or IOSTAT= *scalar-default-int-variable*  
45 or ERR= *label*  
46 or FILE= *file-name-expr*  
47 or STATUS= *scalar-default-char-expr*  
48 or ACCESS= *scalar-default-char-expr*  
49 or FORM= *scalar-default-char-expr*  
50 or RECL= *scalar-int-expr*

- 1 or BLANK= *scalar-default-char-expr*  
 2 or POSITION= *scalar-default-char-expr*  
 3 or ACTION= *scalar-default-char-expr*  
 4 or DELIM= *scalar-default-char-expr*  
 5 or PAD= *scalar-default-char-expr*
- 6 R906 *file-name-expr* is *scalar-default-char-expr*
- 7 Constraint: If the optional characters UNIT= are omitted from the unit specifier, the unit specifier  
 8 must be the first item in the *connect-spec-list*.
- 9 Constraint: Each specifier must not appear more than once in a given *open-stmt*; an *external-file-*  
 10 *unit* must be specified.
- 11 Constraint: The *label* used in the ERR= specifier must be the statement label of a branch target  
 12 statement that appears in the same scoping unit as the OPEN statement.
- 13 If the STATUS= specifier has the value OLD, NEW, or REPLACE, the FILE= specifier must be pre-  
 14 sent. If the STATUS= specifier has the value SCRATCH, the FILE= specifier must be absent.
- 15 A specifier that requires a *scalar-default-char-expr* may have a limited list of character values. These  
 16 values are listed for each such specifier. Any trailing blanks are ignored. If a processor is capable  
 17 of representing letters in both upper and lower case, the value specified is without regard to case.  
 18 Some specifiers have a default value if the specifier is omitted.
- 19 The IOSTAT= specifier and ERR= specifier are described in 9.4.1.4 and 9.4.1.5, respectively.
- 20 An example of an OPEN statement is:
- 21 OPEN (10, FILE = 'employee.names', ACTION = 'READ', PAD = 'YES')
- 22 **9.3.4.1 FILE= Specifier in the OPEN Statement.** The value of the FILE= specifier is the name of  
 23 the file to be connected to the specified unit. Any trailing blanks are ignored. The *file-name-expr*  
 24 must be a name that is allowed by the processor. If this specifier is omitted and the unit is not con-  
 25 nected to a file, the STATUS= specifier must be specified with a value of SCRATCH; in this case,  
 26 the connection is made to a processor-dependent file. If a processor is capable of representing let-  
 27 ters in both upper and lower case, the interpretation of case is processor dependent.
- 28 **9.3.4.2 STATUS= Specifier in the OPEN Statement.** The *scalar-default-char-expr* must evaluate to  
 29 OLD, NEW, SCRATCH, REPLACE, or UNKNOWN. If OLD is specified, the file must exist. If  
 30 NEW is specified, the file must not exist.
- 31 Successful execution of an OPEN statement with NEW specified creates the file and changes the  
 32 status to OLD. If REPLACE is specified and the file does not already exist, the file is created and  
 33 the status is changed to OLD. If REPLACE is specified and the file does exist, the file is deleted, a  
 34 new file is created with the same name, and the status is changed to OLD. If SCRATCH is speci-  
 35 fied, the file is created and connected to the specified unit for use by the executable program but is  
 36 deleted at the execution of a CLOSE statement referring to the same unit or at the termination of  
 37 the executable program. Note that SCRATCH must not be specified with a named file. If  
 38 UNKNOWN is specified, the status is processor dependent. If this specifier is omitted, the default  
 39 value is UNKNOWN.
- 40 **9.3.4.3 ACCESS= Specifier in the OPEN Statement.** The *scalar-default-char-expr* must evaluate to  
 41 SEQUENTIAL or DIRECT. The ACCESS= specifier specifies the access method for the connection  
 42 of the file as being sequential or direct. If this specifier is omitted, the default value is SEQUEN-  
 43 TIAL. For an existing file, the specified access method must be included in the set of allowed  
 44 access methods for the file. For a new file, the processor creates the file with a set of allowed access  
 45 methods that includes the specified method.

## Input/Output Statements

- 1 **9.3.4.4 FORM= Specifier in the OPEN Statement.** The *scalar-default-char-expr* must evaluate to  
2 FORMATTED or UNFORMATTED. The FORM= specifier determines whether the file is being  
3 connected for formatted or unformatted input/output. If this specifier is omitted, the default  
4 value is UNFORMATTED if the file is being connected for direct access, and the default value is  
5 FORMATTED if the file is being connected for sequential access. For an existing file, the specified  
6 form must be included in the set of allowed forms for the file. For a new file, the processor creates  
7 the file with a set of allowed forms that includes the specified form.
- 8 **9.3.4.5 RECL= Specifier in the OPEN Statement.** The value of the RECL= specifier must be posi-  
9 tive. It specifies the length of each record in a file being connected for direct access, or specifies the  
10 maximum length of a record in a file being connected for sequential access. This specifier must be  
11 present when a file is being connected for direct access. If this specifier is omitted when a file is  
12 being connected for sequential access, the default value is processor dependent. If the file is being  
13 connected for formatted input/output, the length is the number of characters for all records that  
14 contain only characters of type default character. If the file is being connected for unformatted  
15 input/output, the length is measured in processor-dependent units. For an existing file, the value  
16 of the RECL= specifier must be included in the set of allowed record lengths for the file. For a new  
17 file, the processor creates the file with a set of allowed record lengths that includes the specified  
18 value.
- 19 **9.3.4.6 BLANK= Specifier in the OPEN Statement.** The *scalar-default-char-expr* must evaluate to  
20 NULL or ZERO. The BLANK= specifier is permitted only for a file being connected for formatted  
21 input/output. If NULL is specified, all blank characters in numeric formatted input fields on the  
22 specified unit are ignored, except that a field of all blanks has a value of zero. If ZERO is specified,  
23 all blanks other than leading blanks are treated as zeros. If this specifier is omitted, the default  
24 value is NULL.
- 25 **9.3.4.7 POSITION= Specifier in the OPEN Statement.** The *scalar-default-char-expr* must evaluate  
26 to ASIS, REWIND, or APPEND. The connection must be for sequential access. A file that did not  
27 exist previously (a new file, either specified explicitly or by default) is positioned at its initial point.  
28 REWIND positions an existing file at its initial point. APPEND positions an existing file such that  
29 the endfile record is the next record, if it has one. If an existing file does not have an endfile  
30 record, APPEND positions the file at its terminal point. ASIS leaves the position unchanged if the  
31 file exists and already is connected. ASIS leaves the position unspecified if the file exists but is not  
32 connected. If this specifier is omitted, the default value is ASIS.
- 33 **9.3.4.8 ACTION= Specifier in the OPEN Statement.** The *scalar-default-char-expr* must evaluate to  
34 READ, WRITE, or READWRITE. READ specifies that the WRITE, PRINT, and ENDFILE state-  
35 ments must not refer to this connection. WRITE specifies that READ statements must not refer to  
36 this connection. READWRITE permits any I/O statements to refer to this connection. If this speci-  
37 fier is omitted, the default value is processor dependent. If READWRITE is included in the set of  
38 allowable actions for a file, both READ and WRITE also must be included in the set of allowed  
39 actions for that file. For an existing file, the specified action must be included in the set of allowed  
40 actions for the file. For a new file, the processor creates the file with a set of allowed actions that  
41 includes the specified action.
- 42 **9.3.4.9 DELIM= Specifier in the OPEN Statement.** The *scalar-default-char-expr* must evaluate to  
43 APOSTROPHE, QUOTE, or NONE. If APOSTROPHE is specified, the apostrophe will be used to  
44 delimit character constants written with list-directed or namelist formatting and all internal apos-  
45 trophe will be doubled. If QUOTE is specified, the quotation mark will be used to delimit charac-  
46 ter constants written with list-directed or namelist formatting and all internal quotation marks will  
47 be doubled. If APOSTROPHE or QUOTE is specified, a *kind-param* and underscore will be used to  
48 precede the leading delimiter of a nondefault character constant. If the value of this specifier is  
49 NONE, a character constant when written will not be delimited by apostrophes or quotation  
50 marks, nor will any internal apostrophes or quotation marks be doubled. If this specifier is omit-  
51 ted, the default value is NONE. This specifier is permitted only for a file being connected for for-  
52 matted input/output. This specifier is ignored during input of a formatted record.



1 **9.3.4.10 PAD= Specifier in the OPEN Statement.** The *scalar-default-char-expr* must evaluate to  
 2 YES or NO. If YES is specified, a formatted input record is padded with blanks (9.4.4.4.2) when an  
 3 input list is specified and the format specification requires more data from a record than the record  
 4 contains. If NO is specified, the input list and the format specification must not require more char-  
 5 acters from a record than the record contains. If this specifier is omitted, the default value is YES.  
 6 This specifier is permitted only for a file being connected for formatted input/output. This speci-  
 7 fier is ignored during output of a formatted record.

8 Note that for nondefault character types, the blank padding character is processor dependent.

9 **9.3.5 The CLOSE Statement.** The CLOSE statement is used to terminate the connection of a  
 10 specified unit to an external file.

11 Execution of a CLOSE statement that refers to a unit may occur in any program unit of an execut-  
 12 able program and need not occur in the same program unit as the execution of an OPEN statement  
 13 referring to that unit.

14 Execution of a CLOSE statement specifying a unit that does not exist or has no file connected to it  
 15 is permitted and affects no file.

16 After a unit has been disconnected by execution of a CLOSE statement, it may be connected again  
 17 within the same executable program, either to the same file or to a different file. After a named file  
 18 has been disconnected by execution of a CLOSE statement, it may be connected again within the  
 19 same executable program, either to the same unit or to a different unit, provided that the file still  
 20 exists.

21 At termination of execution of an executable program for reasons other than an error condition, all  
 22 units that are connected are closed. Each unit is closed with status KEEP unless the file status prior  
 23 to termination of execution was SCRATCH, in which case the unit is closed with status DELETE.  
 24 Note that the effect is as though a CLOSE statement without a STATUS= specifier were executed  
 25 on each connected unit.

26 R907 *close-stmt* is CLOSE ( *close-spec-list* )  
 27 R908 *close-spec* is [ UNIT= ] *external-file-unit*  
 28 or IOSTAT= *scalar-default-int-variable*  
 29 or ERR= *label*  
 30 or STATUS= *scalar-default-char-expr*

31 Constraint: If the optional characters UNIT= are omitted from the unit specifier, the unit specifier  
 32 must be the first item in the *close-spec-list*.

33 Constraint: Each specifier must not appear more than once in a given *close-stmt*; an *external-file-*  
 34 *unit* must be specified.

35 Constraint: The *label* used in the ERR= specifier must be the statement label of a branch target  
 36 statement that appears in the same scoping unit as the CLOSE statement.

37 The *scalar-default-char-expr* has a limited list of character values. Any trailing blanks are ignored. If  
 38 a processor is capable of representing letters in both upper and lower case, the value specified is  
 39 without regard to case.

40 The IOSTAT= specifier and ERR= specifier are described in 9.4.1.4 and 9.4.1.5, respectively.

41 An example of a CLOSE statement is:

42 CLOSE (10, STATUS = 'KEEP')

43 **9.3.5.1 STATUS= Specifier in the CLOSE Statement.** The *scalar-default-char-expr* must evaluate to  
 44 KEEP or DELETE. The STATUS= specifier determines the disposition of the file that is connected  
 45 to the specified unit. KEEP must not be specified for a file whose status prior to execution of a  
 46 CLOSE statement is SCRATCH. If KEEP is specified for a file that exists, the file continues to exist  
 47 after the execution of a CLOSE statement. If KEEP is specified for a file that does not exist, the file  
 48 will not exist after the execution of a CLOSE statement. If DELETE is specified, the file will not

## Input/Output Statements

1 exist after the execution of a CLOSE statement. If this specifier is omitted, the default value is  
2 KEEP, unless the file status prior to execution of the CLOSE statement is SCRATCH, in which case  
3 the default value is DELETE.

4 **9.4 Data Transfer Statements.** The READ statement is the data transfer input statement.  
5 The WRITE statement and the PRINT statement are the data transfer output statements.

6 R909 *read-stmt* is READ ( *io-control-spec-list* ) [ *input-item-list* ]  
7 or READ format [ , *input-item-list* ]  
8 R910 *write-stmt* is WRITE ( *io-control-spec-list* ) [ *output-item-list* ]  
9 R911 *print-stmt* is PRINT format [ , *output-item-list* ]

10 Examples of data transfer statements are:

```
11 READ (6, *) SIZE  
12 READ 10, A, B  
13 WRITE (6, 10) A, S, J  
14 PRINT 10, A, S, J  
15 10 FORMAT (2E16.3, I5)
```

16 **9.4.1 Control Information List.** The *io-control-spec-list* is a control information list that includes:

- 17 (1) A reference to the source or destination of the data to be transferred
- 18 (2) Optional specification of editing processes
- 19 (3) Optional specification to identify a record
- 20 (4) Optional specification of exception handling
- 21 (5) Optional return of count of null values
- 22 (6) Optional return of status
- 23 (7) Optional record advancing specification
- 24 (8) Optional return of number of characters read

25 The control information list governs the data transfer.

26 R912 *io-control-spec* is [ UNIT= ] *io-unit*  
27 or [ FMT= ] *format*  
28 or [ NML= ] *namelist-group-name*  
29 or REC= *scalar-int-expr*  
30 or IOSTAT= *scalar-default-int-variable*  
31 or ERR= *label*  
32 or END= *label*  
33 or ADVANCE= *scalar-default-char-expr*  
34 or SIZE= *scalar-default-int-variable*  
35 or EOR= *label*

36 **Constraint:** An *io-control-spec-list* must contain exactly one *io-unit* and may contain at most one of  
37 each of the other specifiers.

38 **Constraint:** An END=, EOR=, or SIZE= specifier must not appear in a *write-stmt*.

39 **Constraint:** The *label* in the ERR=, EOR=, or END= specifier must be the statement label of a  
40 branch target statement that appears in the same scoping unit as the data transfer  
41 statement.

42 **Constraint:** A *namelist-group-name* must not be present if an *input-item-list* or an *output-item-list* is  
43 present in the data transfer statement.

44 **Constraint:** An *io-control-spec-list* must not contain both a *format* and a *namelist-group-name*.

- 1 Constraint: If the optional characters UNIT= are omitted from the unit specifier, the unit specifier  
2 must be the first item in the control information list.
- 3 Constraint: If the optional characters FMT= are omitted from the format specifier, the format  
4 specifier must be the second item in the control information list and the first item  
5 must be the unit specifier without the optional characters UNIT=.
- 6 Constraint: If the optional characters NML= are omitted from the namelist specifier, the namelist  
7 specifier must be the second item in the control information list and the first item  
8 must be the unit specifier without the optional characters UNIT=.
- 9 Constraint: If the unit specifier specifies an internal file, the *io-control-spec-list* must not contain a  
10 REC= specifier or a *namelist-group-name*.
- 11 Constraint: If the REC= specifier is present, an END= specifier must not appear, a *namelist-*  
12 *group-name* must not appear, and the *format*, if any, must not be an asterisk specifying  
13 list-directed input/output.
- 14 Constraint: An ADVANCE= specifier may be present only in a formatted sequential  
15 input/output statement with explicit format specification (10.1) whose control infor-  
16 mation list does not contain an internal file unit specifier.
- 17 Constraint: If an EOR= specifier is present, and <sup>o</sup>ADVANCE= specifier also must appear.
- 18 A SIZE= specifier may be present only in an input statement that contains an ADVANCE= speci-  
19 fier with the value NO.
- 20 An EOR= specifier may be present only in an input statement that contains an ADVANCE= speci-  
21 fier with the value NO.
- 22 If the data transfer statement contains a *format* or *namelist-group-name*, the statement is a **formatted**  
23 **input/output statement**; otherwise, it is an **unformatted input/output statement**.
- 24 In a data transfer statement, the variable specified in an IOSTAT= or a SIZE= specifier, if any, must  
25 not be associated with any entity in the data transfer input/output list (9.4.2) or *namelist-group-*  
26 *object-list*, nor with a *do-variable* of an *io-implied-do* in the data transfer input/output list.
- 27 In a data transfer statement, if a variable specified in an IOSTAT= or a SIZE= specifier is an array  
28 element reference, its subscript values must not be affected by the data transfer, the *io-implied-do*  
29 processing, or the definition or evaluation of any other specifier in the *io-control-spec-list*.
- 30 For the ADVANCE= specifier, the *scalar-default-char-expr* has a limited list of character values. Any  
31 trailing blanks are ignored. If a processor is capable of representing letters in both upper and  
32 lower case, the value specified is without regard to case.

33 An example of a READ statement is:

34 READ (IOSTAT = IOS, UNIT = 6, FMT = '(10F8.2)') A, B

#### 35 9.4.1.1 Format Specifier.

36 R913 *format* ..... is *default-char-expr*  
37 ..... or *label*  
38 ..... or \*  
39 ..... or *scalar-default-int-variable*

- 40 Constraint: The *label* must be the label of a FORMAT statement that appears in the same scoping  
41 unit as the statement containing the format specifier.
- 42 The *scalar-default-int-variable* must have been assigned (8.2.4) the statement label of a FORMAT statement that appears in the  
43 same scoping unit as the *format*.
- 44 The *default-char-expr* must evaluate to a valid format specification (10.1.1 and 10.1.2). Note that  
45 *default-char-expr* includes a character constant.
- 46 If *default-char-expr* is an array, it is treated as if all of the elements of the array were specified in  
47 array element order and were concatenated.

## Input/Output Statements

1 If *format* is \*, the statement is a **list-directed input/output statement**.

2 An example in which the format is a character expression is:

```
3 READ (6, FMT = "(" // CHAR_FMT // ")" ) X, Y, Z
```

4 where CHAR\_FMT is a default character variable.

5 **9.4.1.2 Namelist Specifier.** The NML= specifier supplies the *namelist-group-name* (5.4). This  
6 name identifies a specific collection of data objects on which transfer is to be performed.

7 If a *namelist-group-name* is present, the statement is a **namelist input/output statement**.

8 **9.4.1.3 Record Number.** The REC= specifier specifies the number of the record that is to be read  
9 or written. This specifier may be present only in an input/output statement that specifies a unit  
10 connected for direct access. If the control information list contains a REC= specifier, the statement  
11 is a **direct access input/output statement**; otherwise, it is a **sequential access input/output state-**  
12 **ment**.

13 **9.4.1.4 Input/Output Status.** Execution of an input/output statement containing the IOSTAT=  
14 specifier causes the variable specified in the IOSTAT= specifier to become defined:

- 15 (1) With a zero value if neither an error condition, an end-of-file condition, nor an end-of-  
16 record condition occurs
- 17 (2) With a processor-dependent positive integer value if an error condition occurs,
- 18 (3) With a processor-dependent negative integer value if an end-of-file condition occurs  
19 and no error condition occurs, or
- 20 (4) With a processor-dependent negative integer value different from the end-of-file value  
21 if an end-of-record condition occurs and no error condition or end-of-file condition  
22 occurs.

23 Note that an end-of-file condition may occur only during execution of a sequential input statement  
24 and an end-of-record condition may occur only during execution of a nonadvancing input state-  
25 ment.

26 Consider the example:

```
27 READ (FMT = "(E8.3)", UNIT=3, IOSTAT = IOSS) X  
28 !  
29 IF (IOSS < 0) THEN  
30 !  
31 ! Perform end-of-file processing on the file  
32 ! connected to unit 3.  
33 CALL END_PROCESSING  
34 !  
35 ELSE IF (IOSS > 0) THEN  
36 !  
37 ! Perform error processing  
38 CALL ERROR_PROCESSING  
39 !  
40 END IF
```

41 **9.4.1.5 Error Branch.** If an input/output statement contains an ERR= specifier and an error con-  
42 dition (9.4.3) occurs during execution of the statement:

- 43 (1) Execution of the input/output statement terminates,
- 44 (2) The position of the file specified in the input/output statement becomes indeterminate,
- 45 (3) If the input/output statement also contains an IOSTAT= specifier, the variable specified  
46 becomes defined with a processor-dependent positive integer value,

- 1           (4) If the statement is a READ statement and it contains a SIZE= specifier, the variable  
2           becomes defined with an integer value (9.4.1.9), and
- 3           (5) Execution continues with the statement specified in the ERR= specifier.
- 4   **9.4.1.6 End-of-File Branch.** If an input statement contains an END= specifier and an end-of-file  
5   condition (9.4.3) occurs and no error condition (9.4.3) occurs during execution of the statement:
- 6           (1) Execution of the input statement terminates,
- 7           (2) If the file specified in the input statement is an external file, it is positioned after the  
8           endfile record.
- 9           (3) If the input statement also contains an IOSTAT= specifier, the variable specified  
10          becomes defined with a processor-dependent negative integer value, and
- 11          (4) Execution continues with the statement specified in the END= specifier.
- 12   In a WRITE statement, the control information list must not contain an END= specifier.
- 13   **9.4.1.7 End-of-Record Branch.** If an input statement contains an EOR= specifier and an end-of-  
14   record condition (9.4.3) occurs and no error condition (9.4.3) occurs during execution of the state-  
15   ment:
- 16          (1) If the PAD= specifier has the value YES, the record is padded with blanks to satisfy the  
17          input list item (9.4.4.2) and corresponding data edit descriptor that requires more  
18          characters than the record contains,
- 19          (2) Execution of the input statement terminates,
- 20          (3) The file specified in the input statement is positioned after the current record,
- 21          (4) If the input statement also contains an IOSTAT= specifier, the variable specified  
22          becomes defined with a processor-dependent negative integer value,
- 23          (5) If the input statement contains a SIZE= specifier, the variable becomes defined with an  
24          integer value (9.4.1.9), and
- 25          (6) Execution continues with the statement specified in the EOR= specifier.
- 26   In a WRITE statement, the control information list must not contain an EOR= specifier.
- 27   **9.4.1.8 Advance Specifier.** The *scalar-default-char-expr* must evaluate to YES or NO. The  
28   ADVANCE= specifier determines whether nonadvancing input/output occurs for this  
29   input/output statement. If NO is specified, nonadvancing input/output occurs. If YES is speci-  
30   fied, advancing formatted sequential input/output occurs. If this specifier is omitted, the default  
31   value is YES.
- 32   **9.4.1.9 Character Count.** When a nonadvancing input statement terminates, the variable speci-  
33   fied in the SIZE= specifier becomes defined with the count of the characters transferred by data  
34   edit descriptors during execution of the current input statement. Blanks inserted as padding  
35   (9.4.4.2) are not counted.
- 36   **9.4.2 Data Transfer Input/Output List.** An input/output list specifies the entities whose values  
37   are transferred by a data transfer input/output statement.
- 38   R914   *input-item*                            is *variable*  
39                                                    or *io-implied-do*
- 40   R915   *output-item*                         is *expr*  
41                                                    or *io-implied-do*
- 42   R916   *io-implied-do*                        is ( *io-implied-do-object-list* , *io-implied-do-control* )
- 43   R917   *io-implied-do-object*                is *input-item*  
44                                                    or *output-item*

## Input/Output Statements

- 1 R918 *io-implied-do-control* is *do-variable* = *scalar-numeric-expr* , ■  
2 ■ *scalar-numeric-expr* [ , *scalar-numeric-expr* ]
- 3 Constraint: A *variable* that is an *input-item* must not be an assumed-size array.
- 4 Constraint: The *do-variable* must be a scalar of type integer, default real, or double precision real.
- 5 Constraint: Each *scalar-numeric-expr* in an *io-implied-do-control* must be of type integer, default real,  
6 or double precision real.
- 7 Constraint: In an *input-item-list*, an *io-implied-do-object* must be an *input-item*. In an *output-item-*  
8 *list*, an *io-implied-do-object* must be an *output-item*.
- 9 An *input-item* must not appear as, nor be associated with, the *do-variable* of any *io-implied-do* that  
10 contains the *input-item*.
- 11 If an input item is a pointer, it must be currently associated with a definable target and data are  
12 transferred from the file to the associated target. If an output item is a pointer, it must be currently  
13 associated with a target and data are transferred from the target to the file.
- 14 The *do-variable* of an *io-implied-do* that is contained within another *io-implied-do* must not appear as,  
15 nor be associated with, the *do-variable* of the containing *io-implied-do*.
- 16 If an array appears as an input/output list item, it is treated as if the elements, if any, were speci-  
17 fied in array element order (6.2.2.2). However, no element of that array may affect the value of any  
18 expression in the *input-item*, nor may any element appear more than once. For example:
- ```
19 INTEGER A (100) , J (100)  
20 ...  
21 READ *, A (A) ! Not allowed  
22 READ *, A (LBOUND (A) : UBOUND (A)) ! Allowed  
23 READ *, A (J) ! Allowed, provided no two elements  
24 ! of J have the same value  
25 READ *, A (A (1) : A (10)) ! Not allowed
```
- 26 A derived-type object must not appear as an input/output list item if any component ultimately  
27 contained within the object is not accessible within the scoping unit containing the input/output  
28 statement. An example is a structure accessed from a module within which its type is PUBLIC but  
29 its components are PRIVATE.
- 30 If a derived type ultimately contains a pointer component, an object of this type must not appear  
31 as an input item nor as the result of the evaluation of an output list item.
- 32 If a derived-type object appears as an input/output list item in a formatted input/output state-  
33 ment, it is treated as if all of the components of the object were specified in the same order as in the  
34 definition of the derived type.
- 35 An input/output list item of derived type in an unformatted input/output statement is treated as  
36 a single value in a processor-dependent form. Note that, in this case, the appearance of a derived-  
37 type object as an input/output list item is not equivalent to the list of its components.
- 38 For an implied-DO, the loop initialization and execution is the same as for a DO construct (8.1.4.4).
- 39 An input/output list must not contain an item of nondefault character type if the input/output  
40 statement specifies an internal file.
- 41 Note that a constant, an expression involving operators or function references, or an expression  
42 enclosed in parentheses may appear as an output list item but must not appear as an input list  
43 item.
- 44 An example of an output list with an implied-DO is:
- ```
45 WRITE (LP, FMT = ' (10F8.2) ' ) (LOG (A (I)), I = 1, N + 9, K), G
```

- 1 **9.4.3 Error, End-of-Record, and End-of-File Conditions.** The set of input/output error condi-  
2 tions is processor dependent.
- 3 An end-of-record condition occurs when a nonadvancing input statement attempts to transfer  
4 data from a position beyond the end of the current record.
- 5 An end-of-file condition occurs in either of the following cases:
- 6 (1) When an endfile record is encountered during the reading of a file connected for  
7 sequential access.
- 8 (2) When an attempt is made to read a record beyond the end of an internal file.
- 9 An end-of-file condition may occur at the beginning of execution of an input statement. An end-  
10 of-file condition also may occur during execution of a formatted input statement when more than  
11 one record is required by the interaction of the input list and the format.
- 12 If an error condition or an end-of-file condition occurs during execution of an input/output state-  
13 ment, execution of the input/output statement terminates and any implied-DO variables become  
14 undefined. If an error condition occurs during execution of an input/output statement, the posi-  
15 tion of the file becomes indeterminate.
- 16 If an error or end-of-file condition occurs on input, all list items become undefined.
- 17 If an end-of-record condition occurs during execution of a nonadvancing input statement, the fol-  
18 lowing occurs: if the PAD= specifier has the value YES, the record is padded with blanks (9.4.4.4.2)  
19 to satisfy the input list item and corresponding data edit descriptor that require more characters  
20 than the record contains; execution of the input statement terminates; and the file specified in the  
21 input statement is positioned after the current record.
- 22 Execution of the executable program is terminated if an error condition occurs during execution of  
23 an input/output statement that contains neither an IOSTAT= nor an ERR= specifier, or if an end-  
24 of-file condition occurs during execution of a READ statement that contains neither an IOSTAT=  
25 specifier nor an END= specifier, or if an end-of-record condition occurs during execution of a non-  
26 advancing READ statement that contains neither an IOSTAT= specifier nor an EOR= specifier.
- 27 **9.4.4 Execution of a Data Transfer Input/Output Statement.** The effect of executing a data  
28 transfer input/output statement must be as if the following operations were performed in the  
29 order specified:
- 30 (1) Determine the direction of data transfer
- 31 (2) Identify the unit
- 32 (3) Establish the format if one is specified
- 33 (4) Position the file prior to data transfer (9.2.1.3.2)
- 34 (5) Transfer data between the file and the entities specified by the input/output list (if any)  
35 or namelist
- 36 (6) Determine whether an error condition, an end-of-file condition, or an end-of-record  
37 condition has occurred
- 38 (7) Position the file after data transfer (9.2.1.3.3)
- 39 (8) Cause any variables specified in the IOSTAT= and SIZE= specifiers to become defined.
- 40 **9.4.4.1 Direction of Data Transfer.** Execution of a READ statement causes values to be trans-  
41 ferred from a file to the entities specified by the input list, if any, or specified within the file itself  
42 for namelist input. Execution of a WRITE or PRINT statement causes values to be transferred to a  
43 file from the entities specified by the output list and format specification, if any, or by the *namelist-*  
44 *group-name* for namelist output. Execution of a WRITE or PRINT statement for a file that does not  
45 exist creates the file unless an error condition occurs.

## Input/Output Statements

1 **9.4.4.2 Identifying a Unit.** A data transfer input/output statement that contains an input/output  
2 control list includes a unit specifier that identifies an external unit or an internal file. A READ  
3 statement that does not contain an input/output control list specifies a particular processor-  
4 dependent unit, which is the same as the unit identified by \* in a READ statement that contains an  
5 input/output control list. The PRINT statement specifies some other processor-dependent unit,  
6 which is the same as the unit identified by \* in a WRITE statement. Thus, each data transfer  
7 input/output statement identifies an external unit or an internal file.

8 The unit identified by a data transfer input/output statement must be connected to a file when  
9 execution of the statement begins. Note that the file may be preconnected.

10 **9.4.4.3 Establishing a Format.** If the input/output control list contains \* as a format, list-  
11 directed formatting is established. If *namelist-group-name* is present, namelist formatting is estab-  
12 lished. If no *format* or *namelist-group-name* is specified, unformatted data transfer is established.  
13 Otherwise, the format specification identified by the format specifier is established. If the format is  
14 an array, the effect is as if all elements of the array were concatenated in array element order.

15 On output, if an internal file has been specified, a format specification that is in the file or is associ-  
16 ated with the file must not be specified.

17 **9.4.4.4 Data Transfer.** Data are transferred between records and entities specified by the  
18 input/output list or namelist. The list items are processed in the order of the input/output list for  
19 all data transfer input/output statements except namelist formatted data transfer statements. The  
20 next item to be processed in the list is called the *next effective item*. Zero-sized arrays and  
21 implied-DO lists with iteration counts of zero are ignored in determining the next effective item.  
22 The list items for a namelist input statement are processed in the order of the entities specified  
23 within the input records. The list items for a namelist output statement are processed in the order  
24 in which the data objects (variables) are specified in the *namelist-group-object-list*.

25 All values needed to determine which entities are specified by an input/output list item are deter-  
26 mined at the beginning of the processing of that item.

27 All values are transmitted to or from the entities specified by a list item prior to the processing of  
28 any succeeding list item for all data transfer input/output statements. In the example,

29 READ (N) N, X (N)

30 the old value of N identifies the unit, but the new value of N is the subscript of X.

31 All values following the *name=* part of the namelist entity (10.9) within the input records are trans-  
32 mitted to the matching entity specified in the *namelist-group-object-list* prior to processing any suc-  
33 ceeding entity within the input record for namelist input statements. If an entity is specified more  
34 than once within the input record during a namelist formatted data transfer input statement, the  
35 last occurrence of the entity specifies the value or values to be used for that entity.

36 An input list item, or an entity associated with it, must not contain any portion of an established  
37 format specification.

38 If the input/output item is a pointer, data are transferred between the file and the associated tar-  
39 get.

40 If an internal file has been specified, an input/output list item must not be in the file or associated  
41 with the file. Note that the file is a data object.

42 A DO variable becomes defined and its iteration count established at the beginning of processing  
43 of the items that constitute the range of an *io-implied-do*.

44 On output, every entity whose value is to be transferred must be defined.

45 **9.4.4.4.1 Unformatted Data Transfer.** During unformatted data transfer, data are transferred  
46 without editing between the current record and the entities specified by the input/output list.  
47 Exactly one record is read or written.



- 1 Objects of intrinsic or derived types may be transferred by means of an unformatted data transfer  
2 statement.
- 3 On input, the file must be positioned so that the record read is an unformatted record or an endfile  
4 record. The number of values required by the input list must be less than or equal to the number  
5 of values in the record. Each value in the record must be of the same type as the corresponding  
6 entity in the input list, except that one complex value may correspond to two real list entities or  
7 two real values may correspond to one complex list entity. The type parameters of the corre-  
8 sponding entities must be the same. Note that if an entity in the input list is of type character, the  
9 character entity must have the same length and the same kind type parameter as the character  
10 value. Also note that if two real values correspond to one complex entity or one complex value  
11 corresponds to two real entities, all three must have the same kind type parameter value.
- 12 On output to a file connected for direct access, the output list must not specify more values than  
13 can fit into the record. If the file is connected for direct access and the values specified by the out-  
14 put list do not fill the record, the remainder of the record is undefined.
- 15 If the file is connected for sequential access, the record is created with a length sufficient to hold  
16 the values from the output list. This length must be one of the set of allowed record lengths for the  
17 file and must not exceed the value specified in the RECL= specifier, if any, of the OPEN statement  
18 that established the connection.
- 19 If the file is connected for formatted input/output, unformatted data transfer is prohibited.
- 20 The unit specified must be an external unit.
- 21 **9.4.4.2 Formatted Data Transfer.** During formatted data transfer, data are transferred with  
22 editing between the file and the entities specified by the input/output list or by the *namelist-group-*  
23 *name*, if any. Format control is initiated and editing is performed as described in Section 10. The  
24 current record and possibly additional records are read or written.
- 25 Values may be transferred by means of a formatted data transfer statement to or from objects of  
26 intrinsic or derived types. In the latter case, the transfer is in the form of values of intrinsic types  
27 to or from the components of intrinsic types that ultimately comprise these structured objects.
- 28 On input, the file must be positioned so that the record read is a formatted record or an endfile  
29 record.
- 30 If the file is connected for unformatted input/output, formatted data transfer is prohibited.
- 31 During advancing input from a file whose PAD= specifier has the value NO, the input list and for-  
32 mat specification must not require more characters from the record than the record contains.
- 33 During advancing input from a file whose PAD= specifier has the value YES, or during input from  
34 an internal file, blank characters are supplied by the processor if the input list and format specifica-  
35 tion require more characters from the record than the record contains.
- 36 During nonadvancing input from a file whose PAD= specifier has the value NO, an end-of-record  
37 condition (9.4.3) occurs if the input list and format specification require more characters from the  
38 record than the record contains.
- 39 During nonadvancing input from a file whose PAD= specifier has the value YES, an end-of-record  
40 condition occurs and blank characters are supplied by the processor if the input list and format  
41 specification require more characters from the record than the record contains.
- 42 If the file is connected for direct access, the record number is increased by one as each succeeding  
43 record is read or written.
- 44 On output, if the file is connected for direct access or is an internal file and the characters specified  
45 by the output list and format do not fill a record, blank characters are added to fill the record.
- 46 On output, the output list and format specification must not specify more characters for a record  
47 than have been specified by a RECL= specifier in the OPEN statement or the record length of an  
48 internal file.

## Input/Output Statements

- 1 **9.4.4.5 List-Directed Formatting.** If list-directed formatting has been established, editing is per-  
2 formed as described in 10.8.
- 3 **9.4.4.6 Namelist Formatting.** If namelist formatting has been established, editing is performed  
4 as described in 10.9.
- 5 **9.4.5 Printing of Formatted Records.** The transfer of information in a formatted record to cer-  
6 tain devices determined by the processor is called **printing**. If a formatted record is printed, the  
7 first character of the record is not printed. The remaining characters of the record, if any, are  
8 printed in one line beginning at the left margin.
- 9 The first character of such a record must be of default character type and determines vertical spac-  
10 ing as follows:

| Character | Vertical Spacing Before Printing |
|-----------|----------------------------------|
| Blank     | One Line                         |
| 0         | Two Lines                        |
| 1         | To First Line of Next Page       |
| +         | No Advance                       |

- 17 If there are no characters in the record, the vertical spacing is one line and no characters other than  
18 blank are printed in that line.
- 19 The PRINT statement does not imply that printing will occur, and the WRITE statement does not  
20 imply that printing will not occur.

21 **9.4.6 Termination of Data Transfer Statements.** Termination of an input/output data transfer  
22 statement occurs when any of the following conditions are met:

- 23 (1) Format processing encounters a data edit descriptor and there are no remaining ele-  
24 ments in the *input-item-list* or *output-item-list*.
- 25 (2) Unformatted or list-directed data transfer exhausts the *input-item-list* or *output-item-list*.
- 26 (3) Namelist output exhausts the *namelist-group-object-list* or namelist input reaches the end  
27 of a record after having processed a name-value subsequence for every item in the  
28 *namelist-group-object-list*.
- 29 (4) An error condition occurs.
- 30 (5) An end-of-file condition occurs.
- 31 (6) A slash (/) is encountered as a value separator (10.8, 10.9) in the record being read dur-  
32 ing list-directed or namelist input.
- 33 (7) An end-of-record condition occurs during execution of a nonadvancing input state-  
34 ment.

## 35 **9.5 File Positioning Statements.**

36 R919 *backspace-stmt* is BACKSPACE *external-file-unit*  
37 or BACKSPACE ( *position-spec-list* )

38 R920 *endfile-stmt* is ENDFILE *external-file-unit*  
39 or ENDFILE ( *position-spec-list* )

40 R921 *rewind-stmt* is REWIND *external-file-unit*  
41 or REWIND ( *position-spec-list* )

42 A file that is not connected for sequential access must not be referred to by a BACKSPACE, an  
43 ENDFILE, or a REWIND statement.

44 R922 *position-spec* is [ UNIT = ] *external-file-unit*

- 1 or IOSTAT = *scalar-default-int-variable*  
 2 or ERR = *label*
- 3 Constraint: The *label* in the ERR= specifier must be the statement label of a branch target state-  
 4 ment that appears in the same scoping unit as the file positioning statement.
- 5 Constraint: If the optional characters UNIT= are omitted from the unit specifier, the unit specifier  
 6 must be the first item in the *position-spec-list*.
- 7 Constraint: A *position-spec-list* must contain exactly one *external-file-unit* and may contain at most  
 8 one of each of the other specifiers.
- 9 The IOSTAT= and ERR= specifiers are described in 9.4.1.4 and 9.4.1.5, respectively.

10 **9.5.1 BACKSPACE Statement.** Execution of a BACKSPACE statement causes the file connected  
 11 to the specified unit to be positioned before the current record if there is a current record, or before  
 12 the preceding record if there is no current record. If there is no current record and no preceding  
 13 record, the position of the file is not changed. Note that if the preceding record is an endfile  
 14 record, the file is positioned before the endfile record. If a BACKSPACE statement causes the  
 15 implicit writing of an endfile record, the file is positioned before the record that precedes the  
 16 endfile record.

17 Backspacing a file that is connected but does not exist is prohibited.

18 Backspacing over records written using list-directed or namelist formatting is prohibited.

19 An example of a BACKSPACE statement is:

20 BACKSPACE (10, ERR = 20)

21 **9.5.2 ENDFILE Statement.** Execution of an ENDFILE statement writes an endfile record as the  
 22 next record of the file. The file is then positioned after the endfile record which becomes the last  
 23 record of the file. If the file also may be connected for direct access, only those records before the  
 24 endfile record are considered to have been written. Thus, only those records may be read during  
 25 subsequent direct access connections to the file.

26 After execution of an ENDFILE statement, a BACKSPACE or REWIND statement must be used to  
 27 reposition the file prior to execution of any data transfer input/output statement or ENDFILE  
 28 statement.

29 Execution of an ENDFILE statement for a file that is connected but does not exist creates the file  
 30 prior to writing the endfile record.

31 An example of an ENDFILE statement is:

32 ENDFILE K

33 **9.5.3 REWIND Statement.** Execution of a REWIND statement causes the specified file to be posi-  
 34 tioned at its initial point. Note that if the file is already positioned at its initial point, execution of  
 35 this statement has no effect on the position of the file.

36 Execution of a REWIND statement for a file that is connected but does not exist is permitted and  
 37 has no effect.

38 An example of a REWIND statement is:

39 REWIND 10

40 **9.6 File Inquiry.** The INQUIRE statement may be used to inquire about properties of a particu-  
 41 lar named file or of the connection to a particular unit. There are three forms of the INQUIRE  
 42 statement: *inquire by file*, which uses the FILE= specifier, *inquire by unit*, which uses the UNIT=  
 43 specifier, and *inquire by output list*, which uses only the IOLENGTH= specifier. All specifier  
 44 value assignments are performed according to the rules for assignment statements.

## Input/Output Statements

1 An INQUIRE statement may be executed before, while, or after a file is connected to a unit. All  
2 values assigned by an INQUIRE statement are those that are current at the time the statement is  
3 executed.

4 R923 *inquire-stmt* is INQUIRE ( *inquire-spec-list* )  
5 or INQUIRE ( IOLENGTH = *scalar-default-int-variable* ) ■  
6 ■ *output-item-list*

7 Examples of INQUIRE statements are:

8 INQUIRE ( IOLENGTH = IOL) A ( 1:N)  
9 INQUIRE ( UNIT = JOAN, OPENED = LOG\_01, NAMED = LOG\_02, &  
10 FORM = CHAR\_VAR, IOSTAT = IOS)

11 **9.6.1 Inquiry Specifiers.** Unless constrained, the following inquiry specifiers may be used in  
12 either of the inquire by file or inquire by unit forms of the INQUIRE statement:

13 R924 *inquire-spec* is [ UNIT = ] *external-file-unit*  
14 or FILE = *file-name-expr*  
15 or IOSTAT = *scalar-default-int-variable*  
16 or ERR = *label*  
17 or EXIST = *scalar-default-logical-variable*  
18 or OPENED = *scalar-default-logical-variable*  
19 or NUMBER = *scalar-default-int-variable*  
20 or NAMED = *scalar-default-logical-variable*  
21 or NAME = *scalar-default-char-variable*  
22 or ACCESS = *scalar-default-char-variable*  
23 or SEQUENTIAL = *scalar-default-char-variable*  
24 or DIRECT = *scalar-default-char-variable*  
25 or FORM = *scalar-default-char-variable*  
26 or FORMATTED = *scalar-default-char-variable*  
27 or UNFORMATTED = *scalar-default-char-variable*  
28 or RECL = *scalar-default-int-variable*  
29 or NEXTREC = *scalar-default-int-variable*  
30 or BLANK = *scalar-default-char-variable*  
31 or POSITION = *scalar-default-char-variable*  
32 or ACTION = *scalar-default-char-variable*  
33 or READ = *scalar-default-char-variable*  
34 or WRITE = *scalar-default-char-variable*  
35 or READWRITE = *scalar-default-char-variable*  
36 or DELIM = *scalar-default-char-variable*  
37 or PAD = *scalar-default-char-variable*

38 Constraint: An *inquire-spec-list* must contain one FILE= specifier or one UNIT= specifier, but not  
39 both, and at most one of each of the other specifiers.

40 Constraint: In the inquire by unit form of the INQUIRE statement, if the optional characters  
41 UNIT= are omitted from the unit specifier, the unit specifier must be the first item in  
42 the inquire-spec-list.

43 When a returned value of a specifier other than the NAME= specifier is of type character and the  
44 processor is capable of representing letters in both upper and lower case, the value returned is in  
45 upper case.

46 If an error condition occurs during execution of an INQUIRE statement, all of the inquiry specifier  
47 variables become undefined, except for the variable in the IOSTAT= specifier (if any).

48 The IOSTAT= and ERR= specifiers are described in 9.4.1.4 and 9.4.1.5, respectively.

- 1 **9.6.1.1 FILE= Specifier in the INQUIRE Statement.** The value of the *file-name-expr* in the FILE=  
2 specifier specifies the name of the file being inquired about. The named file need not exist or be  
3 connected to a unit. The value of the *file-name-expr* must be of a form acceptable to the processor as  
4 a file name. Any trailing blanks are ignored. If a processor is capable of representing letters in  
5 both upper and lower case, the interpretation of case is processor dependent.
- 6 **9.6.1.2 EXIST= Specifier in the INQUIRE Statement.** Execution of an INQUIRE by file statement  
7 causes the *scalar-default-logical-variable* in the EXIST= specifier to be assigned the value true if there  
8 exists a file with the specified name; otherwise, false is assigned. Execution of an INQUIRE by unit  
9 statement causes true to be assigned if the specified unit exists; otherwise, false is assigned.
- 10 **9.6.1.3 OPENED= Specifier in the INQUIRE Statement.** Execution of an INQUIRE by file state-  
11 ment causes the *scalar-default-logical-variable* in the OPENED= specifier to be assigned the value  
12 true if the file specified is connected to a unit; otherwise, false is assigned. Execution of an  
13 INQUIRE by unit statement causes the *scalar-default-logical-variable* to be assigned the value true if  
14 the specified unit is connected to a file; otherwise, false is assigned.
- 15 **9.6.1.4 NUMBER= Specifier in the INQUIRE Statement.** The *scalar-default-int-variable* in the  
16 NUMBER= specifier is assigned the value of the external unit identifier of the unit that is currently  
17 connected to the file. If there is no unit connected to the file, the value -1 is assigned.
- 18 **9.6.1.5 NAMED= Specifier in the INQUIRE Statement.** The *scalar-default-logical-variable* in the  
19 NAMED= specifier is assigned the value true if the file has a name; otherwise, it is assigned the  
20 value false.
- 21 **9.6.1.6 NAME= Specifier in the INQUIRE Statement.** The *scalar-default-char-variable* in the  
22 NAME= specifier is assigned the value of the name of the file if the file has a name; otherwise, it  
23 becomes undefined. Note that if this specifier appears in an INQUIRE by file statement, its value  
24 is not necessarily the same as the name given in the FILE= specifier. For example, the processor  
25 may return a file name qualified by a user identification. However, the value returned must be  
26 suitable for use as the value of the *file-name-expr* in the FILE= specifier in an OPEN statement. If a  
27 processor is capable of representing letters in both upper and lower case, the case of the characters  
28 assigned to *scalar-default-char-variable* is processor dependent.
- 29 **9.6.1.7 ACCESS= Specifier in the INQUIRE Statement.** The *scalar-default-char-variable* in the  
30 ACCESS= specifier is assigned the value SEQUENTIAL if the file is connected for sequential  
31 access, and DIRECT if the file is connected for direct access. If there is no connection, it is assigned  
32 the value UNDEFINED.
- 33 **9.6.1.8 SEQUENTIAL= Specifier in the INQUIRE Statement.** The *scalar-default-char-variable* in the  
34 SEQUENTIAL= specifier is assigned the value YES if SEQUENTIAL is included in the set of  
35 allowed access methods for the file, NO if SEQUENTIAL is not included in the set of allowed  
36 access methods for the file, and UNKNOWN if the processor is unable to determine whether or  
37 not SEQUENTIAL is included in the set of allowed access methods for the file.
- 38 **9.6.1.9 DIRECT= Specifier in the INQUIRE Statement.** The *scalar-default-char-variable* in the  
39 DIRECT= specifier is assigned the value YES if DIRECT is included in the set of allowed access  
40 methods for the file, NO if DIRECT is not included in the set of allowed access methods for the file,  
41 and UNKNOWN if the processor is unable to determine whether or not DIRECT is included in the  
42 set of allowed access methods for the file.
- 43 **9.6.1.10 FORM= Specifier in the INQUIRE Statement.** The *scalar-default-char-variable* in the  
44 FORM= specifier is assigned the value FORMATTED if the file is connected for formatted  
45 input/output, and is assigned the value UNFORMATTED if the file is connected for unformatted  
46 input/output. If there is no connection, it is assigned the value UNDEFINED.

## Input/Output Statements

- 1 **9.6.1.11 FORMATTED= Specifier in the INQUIRE Statement.** The *scalar-default-char-variable* in the  
2 FORMATTED= specifier is assigned the value YES if FORMATTED is included in the set of  
3 allowed forms for the file, NO if FORMATTED is not included in the set of allowed forms for the  
4 file, and UNKNOWN if the processor is unable to determine whether or not FORMATTED is  
5 included in the set of allowed forms for the file.
- 6 **9.6.1.12 UNFORMATTED= Specifier in the INQUIRE Statement.** The *scalar-default-char-variable* in  
7 the UNFORMATTED= specifier is assigned the value YES if UNFORMATTED is included in the  
8 set of allowed forms for the file, NO if UNFORMATTED is not included in the set of allowed  
9 forms for the file, and UNKNOWN if the processor is unable to determine whether or not  
10 UNFORMATTED is included in the set of allowed forms for the file.
- 11 **9.6.1.13 RECL= Specifier in the INQUIRE Statement.** The *scalar-default-int-variable* in the RECL=  
12 specifier is assigned the value of the maximum record length of the file. If the file is connected for  
13 formatted input/output, the length is the number of characters for all records that contain only  
14 characters of type default character. If the file is connected for unformatted input/output, the  
15 length is measured in processor-dependent units. If there is no connection, the *scalar-default-int-*  
16 *variable* becomes undefined.
- 17 **9.6.1.14 NEXTREC= Specifier in the INQUIRE Statement.** The *scalar-default-int-variable* in the  
18 NEXTREC= specifier is assigned the value  $n + 1$ , where  $n$  is the record number of the last record  
19 read or written on the file connected for direct access. If the file is connected but no records have  
20 been read or written since the connection, the *scalar-default-int-variable* is assigned the value 1. If  
21 the file is not connected for direct access or if the position of the file is indeterminate because of a  
22 previous error condition, the *scalar-default-int-variable* becomes undefined.
- 23 **9.6.1.15 BLANK= Specifier in the INQUIRE Statement.** The *scalar-default-char-variable* in the  
24 BLANK= specifier is assigned the value NULL if null blank control is in effect for the file con-  
25 nected for formatted input/output, and is assigned the value ZERO if zero blank control is in  
26 effect for the file connected for formatted input/output. If there is no connection, or if the connec-  
27 tion is not for formatted input/output, the *scalar-default-char-variable* is assigned the value UNDE-  
28 FINED.
- 29 **9.6.1.16 POSITION= Specifier in the INQUIRE Statement.** The *scalar-default-char-variable* in the  
30 POSITION= specifier is assigned the value REWIND if the file is connected by an OPEN statement  
31 for positioning at its initial point, APPEND if the file is connected for positioning before its endfile  
32 record or at its terminal point, and ASIS if the file is connected without changing its position. If  
33 there is no connection or if the file is connected for direct access, the *scalar-default-char-variable* is  
34 assigned the value UNDEFINED. If the file has been repositioned since the connection, the *scalar-*  
35 *default-char-variable* is assigned a processor-dependent value, which must not be REWIND unless  
36 the file is positioned at its initial point and must not be APPEND unless the file is positioned  
37 before its endfile record or at its terminal point.
- 38 **9.6.1.17 ACTION= Specifier in the INQUIRE Statement.** The *scalar-default-char-variable* in the  
39 ACTION= specifier is assigned the value READ if the file is connected for input only, WRITE if the  
40 file is connected for output only, and READWRITE if it is connected for both input and output. If  
41 there is no connection, the *scalar-default-char-variable* is assigned the value UNDEFINED.
- 42 **9.6.1.18 READ= Specifier in the INQUIRE Statement.** The *scalar-default-char-variable* in the  
43 READ= specifier is assigned the value YES if READ is included in the set of allowed actions for the  
44 file, NO if READ is not included in the set of allowed actions for the file, and UNKNOWN if the  
45 processor is unable to determine whether or not READ is included in the set of allowed actions for  
46 the file.

1 **9.6.1.19 WRITE= Specifier in the INQUIRE Statement.** The *scalar-default-char-variable* in the  
2 WRITE= specifier is assigned the value YES if WRITE is included in the set of allowed actions for  
3 the file, NO if WRITE is not included in the set of allowed actions for the file, and UNKNOWN if  
4 the processor is unable to determine whether or not WRITE is included in the set of allowed  
5 actions for the file.

6 **9.6.1.20 READWRITE= Specifier in the INQUIRE Statement.** The *scalar-default-char-variable* in the  
7 READWRITE= specifier is assigned the value YES if READWRITE is included in the set of allowed  
8 actions for the file, NO if READWRITE is not included in the set of allowed actions for the file, and  
9 UNKNOWN if the processor is unable to determine whether or not READWRITE is included in  
10 the set of allowed actions for the file.

11 **9.6.1.21 DELIM= Specifier in the INQUIRE Statement.** The *scalar-default-char-variable* in the  
12 DELIM= specifier is assigned the value APOSTROPHE if the apostrophe is to be used to delimit  
13 character data written by list-directed or namelist formatting. If the quotation mark is used to  
14 delimit such data, the value QUOTE is assigned. If neither the apostrophe nor the quote is used to  
15 delimit the character data, the value NONE is assigned. If there is no connection or if the connec-  
16 tion is not for formatted input/output, the *scalar-default-char-variable* is assigned the value UNDE-  
17 FINED.

18 **9.6.1.22 PAD= Specifier in the INQUIRE Statement.** The *scalar-default-char-variable* in the PAD=  
19 specifier is assigned the value NO if the connection of the file to the unit included the PAD= speci-  
20 fier and its value was NO. Otherwise, the *scalar-default-char-variable* is assigned the value YES.

21 **9.6.2 Restrictions on Inquiry Specifiers.** A variable that may become defined or undefined as a  
22 result of its use in a specifier in an INQUIRE statement, or any associated entity, must not appear  
23 in another specifier in the same INQUIRE statement.

24 The *inquire-spec-list* in an INQUIRE by file statement must contain exactly one FILE= specifier and  
25 must not contain a UNIT= specifier. The *inquire-spec-list* in an INQUIRE by unit statement must  
26 contain exactly one UNIT= specifier and must not contain a FILE= specifier. The unit specified  
27 need not exist or be connected to a file. If it is connected to a file, the inquiry is being made about  
28 the connection and about the file connected.

29 **9.6.3 Inquire by Output List.** The inquire by output list form of the INQUIRE statement does not  
30 include a FILE= or UNIT= specifier, and includes only an IOLENGTH= specifier and an output  
31 list.

32 The *scalar-default-int-variable* in the IOLENGTH= specifier is assigned the processor-dependent  
33 value that would result from the use of the output list in an unformatted output statement. The  
34 value must be suitable as a RECL= specifier in an OPEN statement that connects a file for unfor-  
35 matted direct access when there are input/output statements with the same input/output list.

36 **9.7 Restrictions on Function References and List Items.** A function reference must not  
37 appear in an expression anywhere in an input/output statement if such a reference causes another  
38 input/output statement to be executed. Note that restrictions in the evaluation of expressions  
39 (7.1.7) prohibit certain side effects.

40 **9.8 Restriction on Input/Output Statements.** If a unit, or a file connected to a unit, does not  
41 have all of the properties required for the execution of certain input/output statements, those  
42 statements must not refer to the unit.

11-11-11  
11-11-11  
11-11-11  
11-11-11  
11-11-11



## 10. INPUT/OUTPUT EDITING

1 A format used in conjunction with an input/output statement provides information that directs  
2 the editing between the internal representation of data and the characters of a sequence of format-  
3 ted records.

4 A format specifier (9.4.1.1) in an input/output statement may refer to a FORMAT statement or to a  
5 character expression that contains a format specification. A format specification provides explicit  
6 editing information. The format specifier also may be an asterisk (\*) which indicates list-directed  
7 formatting (10.8). Instead of a format specifier, a *namelist-group-name* may be specified which indi-  
8 cates namelist formatting (10.9).

9 **10.1 Explicit Format Specification Methods.** Explicit format specification may be given:

- 10 (1) In a FORMAT statement, or  
11 (2) In a character expression.

### 12 10.1.1 FORMAT Statement.

13 R1001 *format-stmt* is FORMAT *format-specification*

14 R1002 *format-specification* is ( [ *format-item-list* ] )

15 Constraint: The *format-stmt* must be labeled.

16 Constraint: The comma used to separate *format-items* in a *format-item-list* may be omitted as fol-  
17 lows:

- 18 (1) Between a P edit descriptor and an immediately following F, E, EN, ES, D,  
19 or G edit descriptor (10.6.5)  
20 (2) Before a slash edit descriptor when the optional repeat specification is not  
21 present (10.6.2)  
22 (3) After a slash edit descriptor  
23 (4) Before or after a colon edit descriptor (10.6.3)

24 Blank characters may precede the initial left parenthesis of the format specification. Additional  
25 blank characters may appear at any point within the format specification, with no effect on the  
26 interpretation of the format specification, except within a character string edit descriptor (10.7.1,  
27 10.7.2).

28 Examples of FORMAT statements are:

29 5 FORMAT (1PE12.4, I10)

30 9 FORMAT (I12, /, ' Dates: ', 2 (2I3, I5))

31 **10.1.2 Character Format Specification.** A character expression used as a format specifier in a  
32 formatted input/output statement must evaluate to a character string whose leading part is a valid  
33 format specification. Note that the format specification begins with a left parenthesis and ends  
34 with a right parenthesis.

35 All character positions up to and including the final right parenthesis of the format specification  
36 must be defined at the time the input/output statement is executed, and must not become  
37 redefined or undefined during the execution of the statement. Character positions, if any, follow-  
38 ing the right parenthesis that ends the format specification need not be defined and may contain  
39 any character data with no effect on the interpretation of the format specification.

40 If the format specifier references a character array, it is treated as if all of the elements of the array  
41 were specified in array element order and were concatenated. However, if a format specifier refer-  
42 ences a character array element, the format specification must be contained entirely within that  
43 array element.

## Input/Output Editing

### 1 10.2 Form of a Format Item List.

2 R1003 *format-item* is [ *r* ] *data-edit-desc*  
3 or *control-edit-desc*  
4 or *char-string-edit-desc*  
5 or [ *r* ] ( *format-item-list* )

6 R1004 *r* is *int-literal-constant*

7 Constraint: *r* must be positive.

8 Constraint: *r* must not have a kind parameter specified for it.

9 The integer literal constant *r* is called a **repeat specification**.

10 **10.2.1 Edit Descriptors.** An edit descriptor is a data edit descriptor, a control edit descriptor, or  
11 a character string edit descriptor.

12 R1005 *data-edit-desc* is I *w* [ . *m* ]  
13 or B *w* [ . *m* ]  
14 or O *w* [ . *m* ]  
15 or Z *w* [ . *m* ]  
16 or F *w* . *d*  
17 or E *w* . *d* [ E *e* ]  
18 or EN *w* . *d* [ E *e* ]  
19 or ES *w* . *d* [ E *e* ]  
20 or G *w* . *d* [ E *e* ]  
21 or L *w*  
22 or A [ *w* ]  
23 or D *w* . *d*

24 R1006 *w* is *int-literal-constant*

25 R1007 *m* is *int-literal-constant*

26 R1008 *d* is *int-literal-constant*

27 R1009 *e* is *int-literal-constant*

28 Constraint: *w* and *e* must be positive.

29 Constraint: *w*, *m*, *d*, and *e* must not have kind parameters specified for them.

30 I, B, O, Z, F, E, EN, ES, G, L, A, and D indicate the manner of editing.

31 R1010 *control-edit-desc* is *position-edit-desc*  
32 or [ *r* ] /  
33 or :  
34 or *sign-edit-desc*  
35 or *k* P  
36 or *blank-interp-edit-desc*

37 R1011 *k* is *signed-int-literal-constant*

38 Constraint: *k* must not have a kind parameter specified for it.

39 R1012 *position-edit-desc* is T *n*  
40 or TL *n*  
41 or TR *n*  
42 or *n* X

43 R1013 *n* is *int-literal-constant*

44 Constraint: *n* must be positive.

45 Constraint: *n* must not have a kind parameter specified for it.

46 R1014 *sign-edit-desc* is S

1 or SP  
 2 or SS  
 3 R1015 *blank-interp-edit-desc* is BN  
 4 or BZ

5 In *kP*, *k* is called the scale factor.

6 T, TL, TR, X, slash, colon, S, SP, SS, P, BN, and BZ indicate the manner of editing.

7 R1016 *char-string-edit-desc* is *char-literal-constant*  
 8 or *c H rep-char [ rep-char ] ...*

9 R1017 *c* is *int-literal-constant*

10 Constraint: *c* must be positive.

11 Constraint: *c* must not have a kind parameter specified for it.

12 Constraint: The *rep-char* in the *cH* form must be of default character type.

13 Constraint: The *char-literal-constant* must not have a kind parameter specified for it.

14 Each *rep-char* in a character string edit descriptor must be one of the characters capable of representation by the processor.

16 The character string edit descriptors provide constant data to be output, and are not valid for input.

18 Within a character literal 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.

21 In the H edit descriptor, *c* specifies the number of characters following the H.

22 If a processor is capable of representing letters in both upper and lower case, the edit descriptors are without regard to case except for the characters following the H in the H edit descriptor and the characters in the character constants.

24 **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.

27 **10.3 Interaction Between Input/Output List and Format.** The beginning of formatted data transfer using a format specification initiates format control (9.4.4.2). Each action of format control depends on information jointly provided by:

30 (1) The next edit descriptor contained in the format specification, and

31 (2) The next effective item in the input/output list, if one exists.

32 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, *not for advancing I/O the position within the current record is unchanged.*

36 Except for a format item preceded by a repeat specification *r*, a format specification is interpreted from left to right.

38 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.

41 To each data edit descriptor interpreted in a format specification, there corresponds one effective item specified by the input/output list (9.4.2), except that an input/output list item of type complex requires the interpretation of two F, E, EN, ES, D, or G edit descriptors. For each control edit descriptor or character edit descriptor, there is no corresponding item specified by the input/output list, and format control communicates information directly with the record.

## Input/Output Editing

1 Whenever format control encounters a data edit descriptor in a format specification, it determines  
2 whether there is a corresponding effective item specified by the input/output list. If there is such  
3 an item, it transmits appropriately edited information between the item and the record, and then  
4 format control proceeds. If there is no such item, format control terminates.

5 If format control encounters a colon edit descriptor in a format specification and another effective  
6 input/output list item is not specified, format control terminates.

7 If format control encounters the rightmost parenthesis of a complete format specification and  
8 another effective input/output list item is not specified, format control terminates. However, if  
9 another effective input/output list item is specified, the file is positioned in a manner identical to  
10 the way it is positioned when a slash edit descriptor is processed (10.6.2). Format control then  
11 reverts to the beginning of the format item terminated by the last preceding right parenthesis. If  
12 there is no such preceding right parenthesis, format control reverts to the first left parenthesis of  
13 the format specification. If any reversion occurs, the reused portion of the format specification  
14 must contain at least one data edit descriptor. If format control reverts to a parenthesis that is pre-  
15 ceded by a repeat specification, the repeat specification is reused. Reversion of format control, of  
16 itself, has no effect on the scale factor (10.6.5.1), the sign control edit descriptors (10.6.4), or the  
17 blank interpretation edit descriptors (10.6.6).

18 Example: The format specification:

19 10 FORMAT (1X, 2(F10.3, I5))

20 with an output list of

21 WRITE (10,10) 10.1, 3, 4.7, 1, 12.4, 5, 5.2, 6

22 produces the same output as the format specification:

23 10 FORMAT (1X, F10.3, I5, F10.3, I5/F10.3, I5, F10.3, I5)

24 **10.4 Positioning by Format Control.** After each data edit descriptor or character string edit  
25 descriptor is processed, the file is positioned after the last character read or written in the current  
26 record.

27 After each T, TL, TR, or X edit descriptor is processed, the file is positioned as described in 10.6.1.

28 After each slash edit descriptor is processed, the file is positioned as described in 10.6.2.

29 If format control reverts as described in 10.3, the file is positioned in a manner identical to the way  
30 it is positioned when a slash edit descriptor is processed (10.6.2).

31 During a read operation, any unprocessed characters of the current record are skipped whenever  
32 the next record is read.

33 **10.5 Data Edit Descriptors.** Data edit descriptors cause the conversion of data to or from its  
34 internal representation. Characters in the record must be of default kind if they correspond to the  
35 value of a numeric, logical, or default character data entity, and must be of nondefault kind if they  
36 correspond to the value of a data entity of nondefault character type. Characters transmitted to a  
37 record as a result of processing a character string edit descriptor must be of default kind. On  
38 input, the specified variable becomes defined unless an error condition, an end-of-file condition, or  
39 an end-of-record condition occurs. On output, the specified expression is evaluated.

40 **10.5.1 Numeric Editing.** The I, B, O, Z, F, E, EN, ES, D, and G edit descriptors may be used to  
41 specify the input/output of integer, real, and complex data. The following general rules apply:

42 (1) On input, leading blanks are not significant. The interpretation of blanks, other than  
43 leading blanks, is determined by a combination of any BLANK= specifier (9.3.4.6), the  
44 default for a preconnected or internal file, and any BN or BZ blank control that is cur-  
45 rently in effect for the unit (10.6.6). Plus signs may be omitted. A field containing only  
46 blanks is considered to be zero.

- 1 (2) On input, with F, E, EN, ES, D, and G editing, a decimal point appearing in the input  
2 field overrides the portion of an edit descriptor that specifies the decimal point location.  
3 The input field may have more digits than the processor uses to approximate the value  
4 of the datum.
- 5 (3) On output with I, F, E, EN, ES, D, and G editing, the representation of a positive or zero  
6 internal value in the field may be prefixed with a plus, as controlled by the S, SP, and SS  
7 edit descriptors or the processor. The representation of a negative internal value in the  
8 field must be prefixed with a minus. However, the processor must not produce a nega-  
9 tive signed zero in a formatted output record.
- 10 (4) On output, the representation is right-justified in the field. If the number of characters  
11 produced by the editing is smaller than the field width, leading blanks are inserted in  
12 the field.
- 13 (5) On output, if the number of characters produced exceeds the field width or if an expo-  
14 nent exceeds its specified length using the *Ew.dEe*, *ENw.dEe*, *ESw.dEe*, or *Gw.dEe* edit  
15 descriptor, the processor must fill the entire field of width *w* with asterisks. However,  
16 the processor must not produce asterisks if the field width is not exceeded when  
17 optional characters are omitted. Note that when an SP edit descriptor is in effect, a plus  
18 is not optional.

19 **10.5.1.1 Integer Editing.** The *Iw*, *Iw.m*, *Bw*, *Bw.m*, *Ow*, *Ow.m*, *Zw*, and *Zw.m* edit descriptors indi-  
20 cate that the field to be edited occupies *w* positions. The specified input/output list item must be  
21 of type integer. The G edit descriptor also may be used to edit integer data (10.5.4.1.1).

22 On input, *m* has no effect.

23 In the input field for the I edit descriptor, the character string must be in the form of an optionally  
24 signed integer constant, except for the interpretation of blanks. For the B, O, and Z edit descrip-  
25 tors, the character string must consist of binary, octal, or hexadecimal digits (R408, R409, R410) in  
26 the respective input field. *L.C. ok in HEX. !!*

27 The output field for the *Iw* edit descriptor consists of zero or more leading blanks followed by a  
28 minus if the value of the internal datum is negative, or an optional plus otherwise, followed by the  
29 magnitude of the internal value in the form of an unsigned integer constant without leading zeros.  
30 Note that an integer constant always consists of at least one digit.

31 The output field for the *Bw*, *Ow*, and *Zw* descriptors consists of zero or more leading blanks fol-  
32 lowed by the internal value in a form identical to the digits of a binary, octal, or hexadecimal con-  
33 stant, respectively, with the same value and without leading zeros. Note that a binary, octal, or  
34 hexadecimal constant always consists of at least one digit.

35 The output field for the *Iw.m*, *Bw.m*, *Ow.m*, and *Zw.m* edit descriptor is the same as for the *Iw*, *Bw*,  
36 *Ow*, and *Zw* edit descriptor, respectively, except that the unsigned integer constant consists of at  
37 least *m* digits. If necessary, sufficient leading zeros are included to achieve the minimum of *m* dig-  
38 its. The value of *m* must not exceed the value of *w*. If *m* is zero and the value of the internal datum  
39 is zero, the output field consists of only blank characters, regardless of the sign control in effect.

40 **10.5.1.2 Real and Complex Editing.** The F, E, EN, ES, and D edit descriptors specify the editing  
41 of real and complex data. An input/output list item corresponding to an F, E, EN, ES, or D edit  
42 descriptor must be real or complex. The G edit descriptor also may be used to edit real and com-  
43 plex data (10.5.4.1.2).

44 If a processor is capable of representing letters in both upper and lower case, a lower-case letter is  
45 equivalent to the corresponding upper-case letter in the exponent in a numeric input field.

46 **10.5.1.2.1 F Editing.** The *Fw.d* edit descriptor indicates that the field occupies *w* positions, the  
47 fractional part of which consists of *d* digits.

48 The input field consists of an optional sign, followed by a string of digits optionally containing a  
49 decimal point, including any blanks interpreted as zeros. The *d* has no effect on input if the input

## Input/Output Editing

1 field contains a decimal point. If the decimal point is omitted, the rightmost  $d$  digits of the string,  
 2 with leading zeros assumed if necessary, are interpreted as the fractional part of the value repre-  
 3 sented. The string of digits may contain more digits than a processor uses to approximate the  
 4 value of the constant. The basic form may be followed by an exponent of one of the following  
 5 forms:

- 6 (1) Explicitly signed integer constant
- 7 (2) E followed by zero or more blanks, followed by an optionally signed integer constant
- 8 (3) D followed by zero or more blanks, followed by an optionally signed integer constant

9 An exponent containing a D is processed identically to an exponent containing an E.

10 Note that if the input field does not contain an exponent, the effect is as if the basic form were fol-  
 11 lowed by an exponent with a value of  $-k$ , where  $k$  is the established scale factor (10.6.5.1).

12 The output field consists of blanks, if necessary, followed by a minus if the internal value is nega-  
 13 tive, or an optional plus otherwise, followed by a string of digits that contains a decimal point and  
 14 represents the magnitude of the internal value, as modified by the established scale factor and  
 15 rounded to  $d$  fractional digits. Leading zeros are not permitted except for an optional zero imme-  
 16 diately to the left of the decimal point if the magnitude of the value in the output field is less than  
 17 one. The optional zero must appear if there would otherwise be no digits in the output field.

18 **10.5.1.2.2 E and D Editing.** The  $Ew.d$ ,  $Dw.d$ , and  $Ew.dEe$  edit descriptors indicate that the external  
 19 field occupies  $w$  positions, the fractional part of which consists of  $d$  digits, unless a scale factor  
 20 greater than one is in effect, and the exponent part consists of  $e$  digits. The  $e$  has no effect on input  
 21 and  $d$  has no effect on input if the input field contains a decimal point.

22 The form and interpretation of the input field is the same as for F editing (10.5.1.2.1).

23 The form of the output field for a scale factor of zero is:

24  $[\pm] [0] . x_1 x_2 \cdots x_d \text{exp}$

25 where:

26  $\pm$  signifies a plus or a minus.  
 27

28  $x_1 x_2 \cdots x_d$  are the  $d$  most significant digits of the datum value after rounding.  
 29

30  $\text{exp}$  is a decimal exponent having one of the following forms:  
 31

| Edit Descriptor | Absolute Value of Exponent     | Form of Exponent                                       |
|-----------------|--------------------------------|--------------------------------------------------------|
| $Ew.d$          | $  \text{exp}   \leq 99$       | $E\pm z_1 z_2$ or $\pm 0 z_1 z_2$                      |
|                 | $99 <   \text{exp}   \leq 999$ | $\pm z_1 z_2 z_3$                                      |
| $Ew.dEe$        | $  \text{exp}   \leq 10^e - 1$ | $E\pm z_1 z_2 \cdots z_e$                              |
| $Dw.d$          | $  \text{exp}   \leq 99$       | $D\pm z_1 z_2$ or $E\pm z_1 z_2$<br>or $\pm 0 z_1 z_2$ |
|                 | $99 <   \text{exp}   \leq 999$ | $\pm z_1 z_2 z_3$                                      |

51 where each  $z$  is a digit.  
 52

1 The sign in the exponent is produced. A plus sign is produced if the exponent value is zero. The  
 2 forms  $Ew.d$  and  $Dw.d$  must not be used if  $|exp| > 999$ .

3 The scale factor  $k$  controls the decimal normalization (10.2.1, 10.6.5.1). If  $-d < k \leq 0$ , the output field  
 4 contains exactly  $|k|$  leading zeros and  $d - |k|$  significant digits after the decimal point. If  
 5  $0 < k < d + 2$ , the output field contains exactly  $k$  significant digits to the left of the decimal point and  
 6  $d - k + 1$  significant digits to the right of the decimal point. Other values of  $k$  are not permitted.

7 **10.5.1.2.3 EN Editing.** The EN edit descriptor produces an output field in the form of a real num-  
 8 ber in engineering notation such that the decimal exponent is divisible by three and the absolute  
 9 value of the significand (4.3.1.2) is greater than or equal to 1 and less than 1000, except when the  
 10 output value is zero. The scale factor has no effect on output.

11 The forms of the edit descriptor are  $ENw.d$  and  $ENw.dEe$  indicating that the external field occupies  
 12  $w$  positions, the fractional part of which consists of  $d$  digits and the exponent part consists of  $e$  dig-  
 13 its.

14 The form and interpretation of the input field is the same as for F editing (10.5.1.2.1).

15 The form of the output field is:

16  $[\pm] yyy.x_1x_2 \cdots x_d exp$

17 where:

18  
 19  $\pm$  signifies a plus or a minus.

20  
 21  $yyy$  are the 1 to 3 decimal digits representative of the most significant digits of the value of  
 22 the datum after rounding ( $yyy$  is an integer such that  $1 \leq yyy < 1000$  or, if the output value is  
 23 zero,  $yyy = 0$ ).

24  
 25  $x_1x_2 \cdots x_d$  are the  $d$  next most significant digits of the value of the datum after rounding.

26  
 27  $exp$  is a decimal exponent, divisible by three, of one of the following forms:

| Edit Descriptor | Absolute Value of Exponent | Form of Exponent               |
|-----------------|----------------------------|--------------------------------|
| $ENw.d$         | $ exp  \leq 99$            | $E\pm z_1z_2$ or $\pm 0z_1z_2$ |
|                 | $99 <  exp  \leq 999$      | $\pm z_1z_2z_3$                |
| $ENw.dEe$       | $ exp  \leq 10^e - 1$      | $E\pm z_1z_2 \cdots z_e$       |

42 where each  $z$  is a digit.

43 The sign in the exponent is produced. A plus sign is produced if the exponent value is zero. The  
 44 form  $ENw.d$  must not be used if  $|exp| > 999$ .

45 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                     |

## Input/Output Editing

1 **10.5.1.2.4 ES Editing.** The ES edit descriptor produces an output field in the form of a real number in scientific notation such that the absolute value of the significand (4.3.1.2) is greater than or equal to 1 and less than 10, except when the output value is zero. The scale factor has no effect on output.

5 The forms of the edit descriptor are  $ESw.d$  and  $ESw.dEe$  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.

7 The form and interpretation of the input field is the same as for F editing (10.5.1.2.1).

8 The form of the output field is:

9  $[\pm] y.x_1x_2 \cdots x_d exp$

10 where:

11  
12  $\pm$  signifies a plus or a minus.

13  
14  $y$  is a decimal digit representative of the most significant digit of the value of the datum after rounding.

15  
16  
17  $x_1x_2 \cdots x_d$  are the  $d$  next most significant digits of the value of the datum after rounding.

18  
19  $exp$  is a decimal exponent having one of the following forms:

| Edit Descriptor | Absolute Value of Exponent | Form of Exponent               |
|-----------------|----------------------------|--------------------------------|
| $ESw.d$         | $ exp  \leq 99$            | $E\pm z_1z_2$ or $\pm 0z_1z_2$ |
|                 | $99 <  exp  \leq 999$      | $\pm z_1z_2z_3$                |
| $ESw.dEe$       | $ exp  \leq 10^e - 1$      | $E\pm z_1z_2 \cdots z_e$       |

34 where each  $z$  is a digit.

35 The sign in the exponent is produced. A plus sign is produced if the exponent value is zero. The form  $ESw.d$  must not be used if  $|exp| > 999$ .

37 Examples:

| Internal Value | Output field Using SS, ES12.3 |
|----------------|-------------------------------|
| 6.421          | 6.421E+00                     |
| -.5            | -5.000E-01                    |
| .00217         | 2.170E-03                     |
| 4721.3         | 4.721E+03                     |

44 **10.5.1.2.5 Complex Editing.** A complex datum consists of a pair of separate real data. The editing of a scalar datum of complex data type is specified by two edit descriptors each of which specifies the editing of real data. The first of the edit descriptors specifies the real part; the second specifies the imaginary part. The two edit descriptors may be different. Control and character string edit descriptors may be processed between the edit descriptor for the real part and the edit descriptor for the imaginary part.



- 1 **10.5.2 Logical Editing.** The *Lw* edit descriptor indicates that the field occupies *w* positions. The  
 2 specified input/output list item must be of type logical. The *G* edit descriptor also may be used to  
 3 edit logical data (10.5.4.2).
- 4 The input field consists of optional blanks, optionally followed by a decimal point, followed by a *T*  
 5 for true or *F* for false. The *T* or *F* may be followed by additional characters in the field. Note that  
 6 the logical constants *.TRUE.* and *.FALSE.* are acceptable input forms. If a processor is capable of  
 7 representing letters in both upper and lower case, a lower-case letter is equivalent to the corre-  
 8 sponding upper-case letter in a logical input field.
- 9 The output field consists of *w - 1* blanks followed by a *T* or *F*, depending on whether the value of  
 10 the internal datum is true or false, respectively.
- 11 **10.5.3 Character Editing.** The *A[w]* edit descriptor is used with an input/output list item of type  
 12 character. The *G* edit descriptor also may be used to edit character data (10.5.4.3). All characters  
 13 transferred and converted under control of one *A* or *G* edit descriptor must have the same kind  
 14 type parameter as the data item in the input/output list.
- 15 If a field width *w* is specified with the *A* edit descriptor, the field consists of *w* characters. If a field  
 16 width *w* is not specified with the *A* edit descriptor, the number of characters in the field is the  
 17 length of the corresponding list item, regardless of the value of the kind type parameter.
- 18 Let *len* be the length of the input/output list item. If the specified field width *w* for *A* input is  
 19 greater than or equal to *len*, the rightmost *len* characters will be taken from the input field. If the  
 20 specified field width *w* is less than *len*, the *w* characters will appear left-justified with *len - w* trail-  
 21 ing blanks in the internal representation.
- 22 If the specified field width *w* for *A* output is greater than *len*, the output field will consist of *w - len*  
 23 blanks followed by the *len* characters from the internal representation. If the specified field width  
 24 *w* is less than or equal to *len*, the output field will consist of the leftmost *w* characters from the  
 25 internal representation.
- 26 Note that for nondefault character types, the blank padding character is processor dependent.
- 27 **10.5.4 Generalized Editing.** The *Gw.d* and *Gw.dEe* edit descriptors are used with an  
 28 input/output list item of any intrinsic type. These edit descriptors indicate that the external field  
 29 occupies *w* positions, the fractional part of which consists of a maximum of *d* digits and the expo-  
 30 nent part consists of *e* digits. When these edit descriptors are used to specify the input/output of  
 31 integer, logical, or character data, *d* and *e* have no effect.
- 32 **10.5.4.1 Generalized Numeric Editing.** When used to specify the input/output of integer, real,  
 33 and complex data, the *Gw.d* and *Gw.dEe* edit descriptors follow the general rules for numeric edit-  
 34 ing (10.5.1). Note that the *Gw.dEe* edit descriptor follows any additional rules for the *Ew.dEe* edit  
 35 descriptor.
- 36 **10.5.4.1.1 Generalized Integer Editing.** When used to specify the input/output of integer data,  
 37 the *Gw.d* and *Gw.dEe* edit descriptors follow the rules for the *Iw* edit descriptor (10.5.1.1).
- 38 **10.5.4.1.2 Generalized Real and Complex Editing.** The form and interpretation of the input  
 39 field is the same as for *F* editing (10.5.1.2.1).
- 40 The method of representation in the output field depends on the magnitude of the datum being  
 41 edited. Let *N* be the magnitude of the internal datum. If  $0 < N < 0.1 - 0.5 \times 10^{-d-1}$  or  $N \geq 10^d - 0.5$ ,  
 42 *Gw.d* output editing is the same as *kPEw.d* output editing and *Gw.dEe* output editing is the same as  
 43 *kPEw.dEe* output editing, where *k* is the scale factor (10.6.5.1) currently in effect. If  
 44  $0.1 - 0.5 \times 10^{-d-1} \leq N < 10^d - 0.5$  or *N* is identically 0, the scale factor has no effect, and the value of  
 45 *N* determines the editing as follows:

| Magnitude of Datum                                                     | Equivalent Conversion  |
|------------------------------------------------------------------------|------------------------|
| $N=0$                                                                  | $F(w-n).(d-1), n('b')$ |
| $0.1 - 0.5 \times 10^{-d-1} \leq N < 1 - 0.5 \times 10^{-d}$           | $F(w-n).d, n('b')$     |
| $1 - 0.5 \times 10^{-d} \leq N < 10 - 0.5 \times 10^{-d+1}$            | $F(w-n).(d-1), n('b')$ |
| $10 - 0.5 \times 10^{-d+1} \leq N < 100 - 0.5 \times 10^{-d+2}$        | $F(w-n).(d-2), n('b')$ |
| .                                                                      | .                      |
| .                                                                      | .                      |
| $10^{d-2} - 0.5 \times 10^{-2} \leq N < 10^{d-1} - 0.5 \times 10^{-1}$ | $F(w-n).1, n('b')$     |
| $10^{d-1} - 0.5 \times 10^{-1} \leq N < 10^d - 0.5$                    | $F(w-n).0, n('b')$     |

16 where  $b$  is a blank.  $n$  is 4 for  $Gw.d$  and  $e + 2$  for  $Gw.dEe$ .

17 Note that the scale factor has no effect unless the magnitude of the datum to be edited is outside  
18 the range that permits effective use of F editing.

19 **10.5.4.2 Generalized Logical Editing.** When used to specify the input/output of logical data,  
20 the  $Gw.d$  and  $Gw.dEe$  edit descriptors follow the rules for the  $Lw$  edit descriptor (10.5.2).

21 **10.5.4.3 Generalized Character Editing.** When used to specify the input/output of character  
22 data, the  $Gw.d$  and  $Gw.dEe$  edit descriptors follow the rules for the  $Aw$  edit descriptor (10.5.3).

23 **10.6 Control Edit Descriptors.** A control edit descriptor does not cause the transfer of data  
24 nor the conversion of data to or from internal representation, but may affect the conversions per-  
25 formed by subsequent data edit descriptors.

26 **10.6.1 Position Editing.** The T, TL, TR, and X edit descriptors specify the position at which the  
27 next character will be transmitted to or from the record. If any character skipped by a T, TL, TR, or  
28 X edit descriptor is of type nondefault character, the result of that position editing is processor  
29 dependent.

30 The position specified by a T edit descriptor may be in either direction from the current position.  
31 On input, this allows portions of a record to be processed more than once, possibly with different  
32 editing.

33 The position specified by an X edit descriptor is forward from the current position. On input, a  
34 position beyond the last character of the record may be specified if no characters are transmitted  
35 from such positions. Note that an  $nX$  edit descriptor has the same effect as a  $TRn$  edit descriptor.

36 On output, a T, TL, TR, or X edit descriptor does not by itself cause characters to be transmitted  
37 and therefore does not by itself affect the length of the record. If characters are transmitted to posi-  
38 tions at or after the position specified by a T, TL, TR, or X edit descriptor, positions skipped and  
39 not previously filled are filled with blanks. The result is as if the entire record were initially filled  
40 with blanks.

41 On output, a character in the record may be replaced. However, a T, TL, TR, or X edit descriptor  
42 never directly causes a character already placed in the record to be replaced. Such edit descriptors  
43 may result in positioning such that subsequent editing causes a replacement.

44 **10.6.1.1 T, TL, and TR Editing.** The left tab limit affects file positioning by the T and TL edit  
45 descriptors. Immediately prior to data transfer, the left tab limit becomes defined as the character  
46 position of the current record. If, during data transfer, the file is positioned to another record, the  
47 left tab limit becomes defined as character position one of that record.

48 The  $Tn$  edit descriptor indicates that the transmission of the next character to or from a record is to  
49 occur at the  $n$ th character position of the record, relative to the left tab limit.

- 1 The  $TLn$  edit descriptor indicates that the transmission of the next character to or from the record  
2 is to occur at the character position  $n$  characters backward from the current position. However, if  
3  $n$  is greater than the difference between the current position and the left tab limit, the  $TLn$  edit  
4 descriptor indicates that the transmission of the next character to or from the record is to occur at  
5 the left tab limit.
- 6 The  $TRn$  edit descriptor indicates that the transmission of the next character to or from the record  
7 is to occur at the character position  $n$  characters forward from the current position.
- 8 Note that  $n$  must be specified and must be greater than zero.
- 9 **10.6.1.2 X Editing.** The  $nX$  edit descriptor indicates that the transmission of the next character to  
10 or from a record is to occur at the position  $n$  characters forward from the current position. Note  
11 that the  $n$  must be specified and must be greater than zero.
- 12 **10.6.2 Slash Editing.** The slash edit descriptor indicates the end of data transfer to or from the  
13 current record.
- 14 On input from a file connected for sequential access, the remaining portion of the current record is  
15 skipped and the file is positioned at the beginning of the next record. This record becomes the cur-  
16 rent record. On output to a file connected for sequential access, a new empty record is created fol-  
17 lowing the current record; this new record then becomes the last and current record of the file and  
18 the file is positioned at the beginning of this new record.
- 19 For a file connected for direct access, the record number is increased by one and the file is posi-  
20 tioned at the beginning of the record that has that record number, if there is such a record, and this  
21 record becomes the current record.
- 22 Note that a record that contains no characters may be written on output. If the file is an internal  
23 file or a file connected for direct access, the record is filled with blank characters. Note also that an  
24 entire record may be skipped on input. The repeat specification is optional on the slash edit  
25 descriptor. If it is not specified, the default value is one.
- 26 **10.6.3 Colon Editing.** The colon edit descriptor terminates format control if there are no more  
27 effective items in the input/output list (9.4.2). The colon edit descriptor has no effect if there are  
28 more effective items in the input/output list.
- 29 **10.6.4 S, SP, and SS Editing.** The S, SP, and SS edit descriptors may be used to control optional  
30 plus characters in numeric output fields. At the beginning of execution of each formatted output  
31 statement, the processor has the option of producing a plus in numeric output fields. If an SP edit  
32 descriptor is encountered in a format specification, the processor must produce a plus in any sub-  
33 sequent position that normally contains an optional plus. If an SS edit descriptor is encountered,  
34 the processor must not produce a plus in any subsequent position that normally contains an  
35 optional plus. If an S edit descriptor is encountered, the option of producing the plus is restored to  
36 the processor.
- 37 The S, SP, and SS edit descriptors affect only I, F, E, EN, ES, D, and G editing during the execution  
38 of an output statement. The S, SP, and SS edit descriptors have no effect during the execution of  
39 an input statement.
- 40 **10.6.5 P Editing.** The  $kP$  edit descriptor sets the value of the scale factor to  $k$ . The scale factor may  
41 affect the editing of numeric quantities.
- 42 **10.6.5.1 Scale Factor.** The value of the scale factor is zero at the beginning of execution of each  
43 input/output statement. It applies to all subsequently interpreted F, E, EN, ES, D, and G edit  
44 descriptors until a P edit descriptor is encountered, and then a new scale factor is established.  
45 Note that reversion of format control (10.3) does not affect the established scale factor.
- 46 The scale factor  $k$  affects the appropriate editing in the following manner:

## Input/Output Editing

- 1 (1) On input, with F, E, EN, ES, D, and G editing (provided that no exponent exists in the  
2 field) and F output editing, the scale factor effect is that the externally represented num-  
3 ber equals the internally represented number multiplied by  $10^k$ .
- 4 (2) On input, with F, E, EN, ES, D, and G editing, the scale factor has no effect if there is an  
5 exponent in the field.
- 6 (3) On output, with E and D editing, the significand (4.3.1.2) part of the quantity to be pro-  
7 duced is multiplied by  $10^k$  and the exponent is reduced by  $k$ .
- 8 (4) On output, with G editing, the effect of the scale factor is suspended unless the magni-  
9 tude of the datum to be edited is outside the range that permits the use of F editing. If  
10 the use of E editing is required, the scale factor has the same effect as with E output  
11 editing.
- 12 (5) On output, with EN and ES editing, the scale factor has no effect.

13 **10.6.6 BN and BZ Editing.** The BN and BZ edit descriptors may be used to specify the interpreta-  
14 tion of blanks, other than leading blanks, in numeric input fields. At the beginning of execution of  
15 each formatted input statement, nonleading blank characters from a file connected by an OPEN  
16 statement are interpreted as zeros or are ignored, depending on the value of the BLANK= specifier  
17 (9.3.4.6) currently in effect for the unit; an internal file is treated as if the file had been opened with  
18 BLANK = NULL. If a BN edit descriptor is encountered in a format specification, all nonleading  
19 blank characters in succeeding numeric input fields are ignored. The effect of ignoring blanks is to  
20 treat the input field as if blanks had been removed, the remaining portion of the field right-  
21 justified, and the blanks replaced as leading blanks. However, a field containing only blanks has  
22 the value zero. If a BZ edit descriptor is encountered in a format specification, all nonleading  
23 blank characters in succeeding numeric input fields are treated as zeros.

24 The BN and BZ edit descriptors affect only I, B, O, Z, F, E, EN, ES, D, and G editing during execu-  
25 tion of an input statement. They have no effect during execution of an output statement.

26 **10.7 Character String Edit Descriptors.** A character string edit descriptor must not be used  
27 on input.

28 **10.7.1 Character Constant Edit Descriptor.** The character constant edit descriptor causes char-  
29 acters to be written from the enclosed characters of the edit descriptor itself, including blanks.  
30 Note that a delimiter is either an apostrophe or quote.

31 For a character constant edit descriptor, the width of the field is the number of characters con-  
32 tained in, but not including, the delimiting characters. Within the field, two consecutive delimiting  
33 characters are counted as a single character.

34 **10.7.2 H Editing.** The *cH* edit descriptor causes character information to be written from the next *c* characters (includ-  
35 ing blanks) following the H of the *cH* edit descriptor in the *format-item-list* itself.

36 **10.8 List-Directed Formatting.** The characters in one or more list-directed records constitute  
37 a sequence of values and value separators. The end of a record has the same effect as a blank char-  
38 acter, unless it is within a character constant. Any sequence of two or more consecutive blanks is  
39 treated as a single blank, unless it is within a character constant.

40 Each value is either a null value or one of the forms:

41 *c*  
42 *r\*c*  
43 *r\**

44 where *c* is a literal constant or a nondelimited character constant and *r* is an unsigned, nonzero,  
45 integer literal constant. The *r\*c* form is equivalent to *r* successive appearances of the constant *c*,  
46 and the *r\** form is equivalent to *r* successive appearances of the null value. Neither of these forms

1 may contain embedded blanks, except where permitted within the constant  $c$ .

2 A value separator is one of the following:

- 3 (1) A comma optionally preceded by one or more contiguous blanks and optionally fol-  
4 lowed by one or more contiguous blanks,
- 5 (2) A slash optionally preceded by one or more contiguous blanks and optionally followed  
6 by one or more contiguous blanks, or
- 7 (3) One or more contiguous blanks between two nonblank values or following the last non-  
8 blank value, where a nonblank value is a constant, an  $r*c$  form, or an  $r*$  form.

9 **10.8.1 List-Directed Input.** Input forms acceptable to edit descriptors for a given type are accept-  
10 able for list-directed formatting, except as noted below. The form of the input value must be  
11 acceptable for the type of the next effective item in the list. Blanks are never used as zeros, and  
12 embedded blanks are not permitted in constants, except within character constants and complex  
13 constants as specified below. Note that the end of a record has the effect of a blank, except when it  
14 appears within a character constant. *A value in the input record is interpreted as if an IN edit descriptor with a suitable value of  $w$  were used.*  
15 When the next effective item is of type real, the input form is that of a numeric input field. A  
16 numeric input field is a field suitable for F editing (10.5.1.2.1) that is assumed to have no fractional  
17 digits unless a decimal point appears within the field.

18 When the next effective item is of type complex, the input form consists of a left parenthesis fol-  
19 lowed by an ordered pair of numeric input fields separated by a comma, and followed by a right  
20 parenthesis. The first numeric input field is the real part of the complex constant and the second is  
21 the imaginary part. Each of the numeric input fields may be preceded or followed by blanks. The  
22 end of a record may occur between the real part and the comma or between the comma and the  
23 imaginary part.

24 When the next effective item is of type logical, the input form must not include slashes, blanks, or  
25 commas among the optional characters permitted for L editing.

26 When the next effective item is of type character, the input form consists of a character literal con-  
27 stant of the same kind as the effective list item. Character constants may be continued from the  
28 end of one record to the beginning of the next record, but the end of record must not occur  
29 between a doubled apostrophe in an apostrophe-delimited constant, nor between a doubled quote  
30 in a quote-delimited constant. The end of the record does not cause a blank or any other character  
31 to become part of the constant. The constant may be continued on as many records as needed.  
32 The characters blank, comma, and slash may appear in default character constants.

33 If the next effective item is of type default character and:

- 34 (1) The character constant does not contain the value separators blank, comma, or slash,  
35 and
- 36 (2) The character constant does not cross a record boundary, and
- 37 (3) The first nonblank character is not a quotation mark or an apostrophe, and
- 38 (4) The leading characters are not numeric followed by an asterisk,

39 the delimiting apostrophes or quotation marks are not required. If the delimiters are omitted, the  
40 character constant is terminated by the first blank, comma, slash, or end of record and apostrophes  
41 and quotation marks within the datum are not to be doubled.

42 Let  $len$  be the length of the next effective item, and let  $w$  be the length of the character constant. If  
43  $len$  is less than or equal to  $w$ , the leftmost  $len$  characters of the constant are transmitted to the next  
44 effective item. If  $len$  is greater than  $w$ , the constant is transmitted to the leftmost  $w$  characters of the  
45 next effective item and the remaining  $len - w$  characters of the next effective item are filled with  
46 blanks. Note that the effect is as though the constant were assigned to the next effective item in a  
47 character assignment statement (7.5.1.4).

## Input/Output Editing

1 **10.8.1.1 Null Values.** A null value is specified by:

2 (1) The  $r^*$  form,

3 (2) No characters between consecutive value separators, or

4 (3) No characters before the first value separator in the first record read by each execution  
5 of a list-directed input statement.

6 Note that the end of a record following any other value separator, with or without separating  
7 blanks, does not specify a null value. A null value has no effect on the definition status of the next  
8 effective item. A null value must not be used for either the real or imaginary part of a complex  
9 constant, but a single null value may represent an entire complex constant.

10 A slash encountered as a value separator during execution of a list-directed input statement causes  
11 termination of execution of that input statement after the assignment of the previous value. Any  
12 characters remaining in the current record are ignored. If there are additional items in the input  
13 list, the effect is as if null values had been supplied for them. Any implied-DO variable in the  
14 input list is defined as though enough null values had been supplied for any remaining input list  
15 items.

16 Note that all blanks in a list-directed input record are considered to be part of some value separa-  
17 tor except for the following:

18 (1) Blanks embedded in a character constant

19 (2) Embedded blanks surrounding the real or imaginary part of a complex constant

20 (3) Leading blanks in the first record read by each execution of a list-directed input state-  
21 ment, unless immediately followed by a slash or comma

### 22 **10.8.1.2 List-Directed Input Example.**

23 INTEGER I; REAL X (8); CHARACTER (11) P;

24 COMPLEX Z; LOGICAL G

25 ...

26 READ \*, I, X, P, Z, G

27 ...

28 The input data records are:

29 12345,12345,,2\*1.5,4\*

30 ISN'T\_BOB'S,(123,0),.TEXAS\$

31 The results are:

| Variable      | Value       |
|---------------|-------------|
| I             | 12345       |
| X (1)         | 12345.0     |
| X (2)         | unchanged   |
| X (3)         | 1.5         |
| X (4)         | 1.5         |
| X (5) - X (8) | unchanged   |
| P             | ISN'T_BOB'S |
| Z             | (123.0,0.0) |
| G             | true        |

43 **10.8.2 List-Directed Output.** The form of the values produced is the same as that required for  
44 input, except as noted otherwise. With the exception of nondelimited character constants, the val-  
45 ues are separated by one or more blanks or by a comma optionally preceded by one or more  
46 blanks and optionally followed by one or more blanks.

- 1 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.
- 2
- 3
- 4 Logical output constants are T for the value true and F for the value false.
- 5 For numeric output, reasonable processor-dependent values of  $w$ ,  $d$ , and  $e$  are used for each of the
- 6 ~~cases involved.~~ *numeric constants output.*
- 7 Integer output constants are produced with the effect of an  $Iw$  edit descriptor.
- 8 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  $OPFw.d$ ; otherwise,  $1PEw.dEe$  is used.
- 9
- 10
- 11
- 12 Complex constants are enclosed in parentheses with a comma separating the real and imaginary parts, each produced as defined above for real constants. The end of a record may occur between the comma and the imaginary part only if the entire constant is as long as, or longer than, an entire record. The only embedded blanks permitted within a complex constant are between the comma and the end of a record and one blank at the beginning of the next record.
- 13
- 14
- 15
- 16
- 17 Character constants produced for an internal file, or for a file opened without a DELIM= specifier (9.3.4.9) or with a DELIM= specifier with a value of NONE:
- 18
- 19 (1) Are not delimited by apostrophes or quotation marks,
  - 20 (2) Are not preceded or followed by a value separator,
  - 21 (3) Have each internal apostrophe or quotation mark represented externally by one apostrophe or quotation mark, and
  - 22
  - 23 (4) Have a blank character inserted by the processor for carriage control at the beginning of
  - 24 any record that begins with the continuation of a character constant from the preceding
  - 25 record.
- 26 Character constants produced for a file opened with a DELIM= specifier with a value of QUOTE are delimited by quotes, possibly are preceded by a *kind-param* and an underscore, are preceded and followed by a value separator, and have each internal quote represented on the external medium by two contiguous quotes.
- 27
- 28
- 29
- 30 Character constants produced for a file opened with a DELIM= specifier with a value of APOSTROPHE are delimited by apostrophes, possibly are preceded by a *kind-param* and an underscore, are preceded and followed by a value separator, and have each internal apostrophe represented on the external medium by two contiguous apostrophes.
- 31
- 32
- 33
- 34 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.
- 35
- 36
- 37 Slashes, as value separators, and null values are not produced as output by list-directed formatting.
- 38
- 39 Except for continuation of delimited character constants, each output record begins with a blank character to provide carriage control when the record is printed.
- 40
- 41 **10.9 Namelist Formatting.** The characters in one or more namelist records constitute a
- 42 sequence of name-value subsequences, each of which consists of an object name or a subobject
- 43 designator followed by an equals and followed by one or more values and value separators. The
- 44 equals may optionally be preceded or followed by one or more contiguous blanks. The end of a
- 45 record has the same effect as a blank character, unless it is within a character constant. Any
- 46 sequence of two or more consecutive blanks is treated as a single blank, unless it is within a character
- 47 constant.

## Input/Output Editing

1 The name may be any name in the *namelist-group-object-list* (5.4).

2 Each value is either a null value (10.9.1.4) or one of the forms:

3  $c$

4  $r*c$

5  $r*$

6 where  $c$  is a literal constant and  $r$  is an unsigned, nonzero, integer literal constant. The  $r*c$  form is  
7 equivalent to  $r$  successive appearances of the constant  $c$ , and the  $r*$  form is equivalent to  $r$  succes-  
8 sive null values. Neither of these forms may contain embedded blanks, except where permitted  
9 within the constant  $c$ .

10 A value separator for namelist formatting is the same as for list-directed formatting (10.8).

11 **10.9.1 Namelist Input.** Input for a namelist input statement consists of:

12 (1) Optional blanks,

13 (2) The character & followed immediately by the *namelist-group-name* specified in the name-  
14 list input statement,

15 (3) One or more blanks,

16 (4) A sequence of zero or more name-value subsequences separated by value separators,  
17 and

18 (5) A slash to terminate the namelist input statement.

19 In each name-value subsequence, the name must be the name of a namelist group object list item  
20 with an optional qualification and the name with the optional qualification must not be a zero-  
21 sized array, a zero-sized array section, or a zero-length character string.

22 If a processor is capable of representing letters in both upper and lower case, a group name or  
23 object name is without regard to case.

24 **10.9.1.1 Namelist Group Object Names.** Within the input data, each name must correspond to  
25 a specific namelist group object name. Subscripts, strides, and substring range expressions used to  
26 qualify group object names must be optionally signed integer literal constants. If a namelist group  
27 object is an array, the input record corresponding to it may contain either the array name or the  
28 designator of a subobject of that array, using the syntax of subobject designators (R602). If the  
29 namelist group object name is the name of a variable of derived type, the name in the input record  
30 may be either the name of the variable or the designator of one of its components, indicated by  
31 qualifying the variable name with the appropriate component name. Successive qualifications  
32 may be applied as appropriate to the shape and type of the variable represented.

33 The order of names in the input records need not match the order of the namelist group object  
34 items. The input records need not contain all the names of the namelist group object items. The  
35 definition status of any names from the *namelist-group-object-list* that do not occur in the input  
36 record remains unchanged. The name in the input record may be preceded and followed by one  
37 or more optional blanks but must not contain embedded blanks.

38 **10.9.1.2 Namelist Input Values.** The datum  $c$  is any input value acceptable to format specifica-  
39 tions for a given type, except for a restriction on the form of input values corresponding to list  
40 items of type logical as specified in 10.9.1.3. The form of the input value must be acceptable for the  
41 type of the namelist group object list item. The number and forms of the input values that may fol-  
42 low the equals in a name-value subsequence depend on the shape and type of the object repre-  
43 sented by the name in the input record. When the name in the input record is that of a scalar vari-  
44 able of an intrinsic type, the equals must not be followed by more than one value. Blanks are  
45 never used as zeros, and embedded blanks are not permitted in constants except within character  
46 constants and complex constants as specified in 10.9.1.3. *The name value subsequences are*  
47 *A namelist group object name or subobject designator may appear in more than one*  
48 *name-value subsequence.* When the name in the input record represents an array variable or a variable of derived type, the  
effect is as if the variable represented were expanded into a sequence of scalar list items of intrinsic



1 data types, in the same way that formatted input/output list items are expanded (9.4.2). Each  
 2 input value following the equals must then be acceptable to format specifications for the intrinsic  
 3 type of the list item in the corresponding position in the expanded sequence, except as noted. The  
 4 number of values following the equals must not exceed the number of list items in the expanded  
 5 sequence, but may be less; in the latter case, the effect is as if sufficient null values had been  
 6 appended to match any remaining list items in the expanded sequence. For example, if the name  
 7 in the input record is the name of an integer array of size 100, at most 100 values, each of which is  
 8 either a digit string or a null value, may follow the equals; these values would then be assigned to  
 9 the elements of the array in array element order.

10 A slash encountered as a value separator during the execution of a namelist input statement causes  
 11 termination of execution of that input statement after assignment of the previous value. If there  
 12 are additional items in the namelist group object being transferred, the effect is as if null values  
 13 had been supplied for them. )

14 Successive namelist records are read by namelist input until a slash is encountered; the remainder  
 15 of the record is ignored.

16 **10.9.1.3 Namelist Group Object List Items.** When the corresponding namelist group object list  
 17 item is of type real, the input form of the input value is that of a numeric input field. A numeric  
 18 input field is a field suitable for F editing (10.5.1.2.1) that is assumed to have no fractional digits  
 19 unless a decimal point appears within the field.

20 When the ~~corresponding~~ <sup>next effective ✓ 116</sup> list item is of type complex, the input form of the input value consists of a  
 21 left parenthesis followed by an ordered pair of numeric input fields separated by a comma and fol-  
 22 lowed by a right parenthesis. The first numeric input field is the real part of the complex constant  
 23 and the second part is the imaginary part. Each of the numeric input fields may be preceded or  
 24 followed by blanks. The end of a record may occur between the real part and the comma or  
 25 between the comma and the imaginary part.

26 When the ~~corresponding~~ <sup>next effective ✓ 116</sup> list item is of type logical, the input form of the input value must not  
 27 include slashes, blanks, or commas among the optional characters permitted for L editing (10.5.2).

28 When the ~~corresponding~~ <sup>next effective ✓ 116</sup> list item is of type character, the input form consists of a character literal  
 29 constant of the same kind as the corresponding list item. Character constants may be continued  
 30 from the end of one record to the beginning of the next record, but the end of record must not  
 31 occur between a doubled apostrophe in an apostrophe-delimited constant, nor between a doubled  
 32 quote in a quote-delimited constant. The end of the record does not cause a blank or any other  
 33 character to become part of the constant. The constant may be continued on as many records as  
 34 needed. The characters blank, comma, and slash may appear in character constants.

35 Let  $len$  be the length of the list item, and let  $w$  be the length of the character constant. If  $len$  is less  
 36 than or equal to  $w$ , the leftmost  $len$  characters of the constant are transmitted to the list item. If  $len$   
 37 is greater than  $w$ , the constant is transmitted to the leftmost  $w$  characters of the list item and the  
 38 remaining  $len - w$  characters of the list item are filled with blanks. Note that the effect is as though  
 39 the constant were assigned to the list item in a character assignment statement (7.5.1.4).

40 **10.9.1.4 Null Values.** A null value is specified by:

- 41 (1) The  $r^*$  form,
- 42 (2) Blanks between two consecutive value separators following an equals,
- 43 (3) Zero or more blanks preceding the first value separator and following an equals, or
- 44 (4) Two consecutive nonblank value separators.

45 A null value has no effect on the definition status of the corresponding input list item. If the name-  
 46 list group object list item is defined, it retains its previous value; if it is undefined, it remains unde-  
 47 fined. A null value must not be used as either the real or imaginary part of a complex constant,  
 48 but a single null value may represent an entire complex constant.

move  
to 9.4.4.4  
v.s.

## Input/Output Editing

- 1 Note that the end of a record following a value separator, with or without intervening blanks, does  
2 not specify a null value.
- 3 **10.9.1.5 Blanks.** All blanks in a namelist input record are considered to be part of some value  
4 separator except for:
- 5 (1) Blanks embedded in a character constant,  
6 (2) Embedded blanks surrounding the real or imaginary part of a complex constant,  
7 (3) Leading blanks following the equals unless followed immediately by a slash or comma,  
8 and  
9 (4) Blanks between a name and the following equals.

### 10 10.9.1.6 Namelist Input Example.

```
11 INTEGER I; REAL X (8); CHARACTER (11) P; COMPLEX Z;  
12 LOGICAL G  
13 NAMELIST / TODAY / G, I, P, Z, X  
14 READ (*, NML = TODAY)
```

15 The input data records are:

```
16 &TODAY I = 12345, X(1) = 12345, X(3:4) = 2*1.5, I=6,  
17 P = "ISN'T_BOB'S", Z = (123,0)/
```

18 The results stored are:

| Variable      | Value              |
|---------------|--------------------|
| I             | <del>12345</del> 6 |
| X (1)         | 12345.0            |
| X (2)         | unchanged          |
| X (3)         | 1.5                |
| X (4)         | 1.5                |
| X (5) - X (8) | unchanged          |
| P             | ISN'T_BOB'S        |
| Z             | (123.0,0.0)        |
| G             | unchanged          |

30 **10.9.2 Namelist Output.** The form of the output produced is the same as that required for input,  
31 except for the forms of real and logical constants. If the processor is capable of representing letters  
32 in both upper and lower case, the name in the output is in upper case. With the exception of non-  
33 delimited character constants, the values are separated by one or more blanks or by a comma  
34 optionally preceded by one or more blanks and optionally followed by one or more blanks.

35 The processor may begin new records as necessary. However, except for complex constants and  
36 character constants, the end of a record must not occur within a constant or a name, and blanks  
37 must not appear within a constant or a name.

38 **10.9.2.1 Namelist Output Editing.** Logical output constants are T for the value true and F for the  
39 value false.

40 Integer output constants are produced with the effect of an Iw edit descriptor.

41 Real constants are produced with the effect of either an F edit descriptor or an E edit descriptor,  
42 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  
43 processor-dependent integers. If the magnitude  $x$  is within this range, the constant is produced  
44 using OPFw.d; otherwise, 1PEw.dEe is used.

45 For numeric output, reasonable processor-dependent integer values of  $w$ ,  $d$ , and  $e$  are used for each  
46 of the cases involved. *numeric constants output.*

- 1 Complex constants are enclosed in parentheses with a comma separating the real and imaginary  
 2 parts, each produced as defined above for real constants. The end of a record may occur between  
 3 the comma and the imaginary part only if the entire constant is as long as, or longer than, an entire  
 4 record. The only embedded blanks permitted within a complex constant are between the comma  
 5 and the end of a record and one blank at the beginning of the next record.
- 6 Character constants produced for a file opened without a DELIM= specifier (9.3.4.9) or with a  
 7 DELIM= specifier with a value of NONE:
- 8 (1) Are not delimited by apostrophes or quotation marks,
  - 9 (2) Are not preceded or followed by a value separator,
  - 10 (3) Have each internal apostrophe or quotation mark represented externally by one apos-  
 11 trophe or quotation mark, and
  - 12 (4) Have a blank character inserted by the processor for carriage control at the beginning of  
 13 any record that begins with the continuation of a character constant from the preceding  
 14 record.
- 15 Character constants produced for a file opened with a DELIM= specifier with a value of QUOTE  
 16 are delimited by quotes, possibly are preceded by a *kind-param* and an underscore, are preceded  
 17 and followed by a value separator, and have each internal quote represented on the external  
 18 medium by two contiguous quotes.
- 19 Character constants produced for a file opened with a DELIM= specifier with a value of APOS-  
 20 TROPHE are delimited by apostrophes, possibly are preceded by a *kind-param* and an underscore,  
 21 are preceded and followed by a value separator, and have each internal apostrophe represented on  
 22 the external medium by two contiguous apostrophes.
- 23 **10.9.2.2 Namelist Output Records.** If two or more successive values in an array in an output  
 24 record produced have identical values, the processor has the option of producing a repeated con-  
 25 stant of the form *r\*c* instead of the sequence of identical values.
- 26 The name of each namelist group object list item is placed in the output record followed by an  
 27 equals and one or more values of the namelist group object list item; however, if the namelist  
 28 group object list item is a zero-sized array or a zero-length character entity, nothing is placed in the  
 29 output record.
- 30 An ampersand character followed immediately by a *namelist-group-name* will be produced by  
 31 namelist formatting at the start of the first output record to indicate which specific group of data  
 32 objects is being output. A slash is produced by namelist formatting to indicate the end of the  
 33 namelist formatting.
- 34 A null value is not produced by namelist formatting.
- 35 Except for continuation of delimited character constants, each output record begins with a blank  
 36 character to provide carriage control when the record is printed.



## 11. PROGRAM UNITS

1 The terms and basic concepts of program units were introduced in 2.2. A program unit is a main  
2 program, an external subprogram, a module, or a block data program unit.  
3 This section describes all of these program units except external subprograms, which are described  
4 in Section 12.

5 **11.1 Main Program.** A **main program** is a program unit that does not contain a SUBROU-  
6 TINE, FUNCTION, MODULE, or BLOCK DATA statement as its first statement.

7 R1101 *main-program* is [ *program-stmt* ]  
8 [ *specification-part* ]  
9 [ *execution-part* ]  
10 [ *internal-subprogram-part* ]  
11 *end-program-stmt*

12 R1102 *program-stmt* is PROGRAM *program-name*

13 R1103 *end-program-stmt* is END [ PROGRAM [ *program-name* ] ]

14 Constraint: In a *main-program*, the *execution-part* must not contain a RETURN statement or an  
15 ENTRY statement.

16 Constraint: The *program-name* may be included in the *end-program-stmt* only if the optional  
17 *program-stmt* is used and, if included, must be identical to the *program-name* specified  
18 in the *program-stmt*.

19 Constraint: An automatic object must not appear in the *specification-part* (R204) of a main pro-  
20 gram.

21 The **program name** is global to the executable program, and must not be the same as the name of  
22 any other program unit, external procedure, or common block in the executable program, nor the  
23 same as any local name in the main program.

24 An example of a main program is:

```
25 PROGRAM ANALYSE  
26     REAL A, B, C (10,10)      ! Specification part  
27     CALL FIND                ! Execution part  
28 CONTAINS  
29     SUBROUTINE FIND          ! Internal procedure  
30     ...  
31     END SUBROUTINE FIND  
32 END PROGRAM ANALYSE
```

33 **11.1.1 Main Program Specifications.** The specifications in the scoping unit of the main program  
34 must not include an OPTIONAL statement, an INTENT statement, a PUBLIC statement, a PRI-  
35 VATE statement, or their equivalent attributes (5.1.2). A SAVE statement has no effect in a main  
36 program. *an ENTRY statement*

37 **11.1.2 Main Program Executable Part.** The sequence of *execution-part* statements specifies the  
38 actions of the main program during program execution. Execution of an executable program  
39 (R201) begins with the first executable construct of the main program.

40 A main program must not be recursive; that is, a reference to it must not appear in any program  
41 unit in the executable program, including itself.

42 Execution of an executable program ends with execution of the *end-program-stmt* of the main pro-  
43 gram or with execution of a STOP statement in any program unit of the executable program.

## Program Units

- 1 **11.1.3 Main Program Internal Procedures.** Any definitions of internal procedures in the main  
2 program must follow the CONTAINS statement. Internal procedures are described in 12.1.2.2.  
3 The main program is called the host of its internal procedures.

- 4 **11.2 External Subprograms.** External subprograms are described in Section 12.

- 5 **11.3 Modules.** A module contains specifications and definitions that are to be accessible to  
6 other program units.

7 R1104 *module* is *module-stmt*  
8 [ *specification-part* ]  
9 [ *module-subprogram-part* ]  
10 *end-module-stmt*

11 R1105 *module-stmt* is MODULE *module-name*

12 R1106 *end-module-stmt* is END [ MODULE [ *module-name* ] ]

13 Constraint: If the *module-name* is specified in the *end-module-stmt*, it must be identical to the  
14 *module-name* specified in the *module-stmt*.

15 Constraint: A module *specification-part* must not contain a *stmt-function-stmt* or a *format-stmt*.

16 Constraint: An automatic object must not appear in the *specification-part* (R204) of a module. <sup>an entry-stmt</sup>

17 The module name is global to the executable program, and must not be the same as the name of  
18 any other program unit, external procedure, or common block in the executable program, nor be  
19 the same as any local name in the module.

20 Note that although statement function definitions and <sup>ENTRY statements</sup> FORMAT statements must not appear in the  
21 specification part of a module, they may appear in the specification part of a module subprogram  
22 contained in the module.

23 Note that a module is host to any module procedures (12.1.2.2) it contains, and that entities in the  
24 module are therefore accessible in the module procedures through host association.

25 **11.3.1 Module Reference.** A USE statement specifying a module name is a **module reference**.  
26 At the time a module reference is processed, the public portions of the specified module must be  
27 available. A module must not reference itself, either directly or indirectly.

28 The accessibility, public or private, of specifications and definitions in a module to a scoping unit  
29 making reference to the module may be controlled in both the module and the scoping unit mak-  
30 ing the reference. In the module, the PRIVATE statement, the PUBLIC statement (5.2.3), their  
31 equivalent attributes (5.1.2.2), and the PRIVATE statement in a derived-type definition (4.4.1) are  
32 used to control the accessibility of module entities outside the module.

33 In a scoping unit making reference to a module, the ONLY option on the USE statement may be  
34 used to further limit the accessibility, in that referencing scoping unit, of the public entities in the  
35 module.

36 **11.3.2 The USE Statement and Use Association.** The USE statement provides the means by  
37 which a scoping unit accesses named data objects, derived types, interface blocks, procedures,  
38 generic identifiers (12.3.2.1), and namelist groups in a module. The entities in the scoping unit are  
39 said to be **use associated** with the entities in the module. The accessed entities have the attributes  
40 specified in the module.

41 R1107 *use-stmt* is USE *module-name* [ , *rename-list* ]  
42 or USE *module-name* , ONLY : [ *only-list* ]

43 R1108 *rename* is *local-name* => *use-name*

44 R1109 *only* is *access-id*  
45 or [ *local-name* => ] *use-name*

- 1 Constraint: Each *access-id* must be a public entity in the module.
- 2 Constraint: Each *use-name* must be the name of a public entity in the module.
- 3 If a *local-name* appears in a *rename-list* or an *only-list*, it is the local name for the entity specified by  
4 *use-name*; otherwise, the local name is the *use-name*.
- 5 The USE statement without the ONLY option provides access to all public entities in the specified  
6 module.
- 7 A USE statement with the ONLY option provides access only to those entities that appear as  
8 *access-ids* or *use-names* in the *only-list*.
- 9 More than one USE statement for a given module may appear in a scoping unit. If one of the USE  
10 statements is without an ONLY qualifier, all public entities in the module are accessible and the  
11 *rename-lists* and *only-lists* are interpreted as a single concatenated *rename-list*. If all the USE state-  
12 ments have ONLY qualifiers, only those entities named in one or more of the *only-lists* are accessi-  
13 ble, that is, all the *only-lists* are interpreted as a single concatenated *only-list*.
- 14 If two or more generic interfaces that are accessible in a scoping unit have the same name, the  
15 same operator, or are both assignments, they are interpreted as a single generic interface. Two or  
16 more accessible entities, other than generic interfaces, may have the same name only if no entity is  
17 referenced by this name in the scoping unit. Except for these cases, the local name of any entity  
18 given accessibility by a USE statement must differ from the local names of all other entities accessi-  
19 ble to the scoping unit through USE statements and otherwise. Note that an entity may be  
20 accessed by more than one local name. } why  
here?  
move  
to  
12-5:26+  
Add?
- 21 The local name of an entity made accessible by a USE statement may appear in no other specifica-  
22 tion statement that would cause any attribute (5.1.2) of the entity to be respecified in the scoping  
23 unit that contains the USE statement, except that it may appear in a PUBLIC or PRIVATE state-  
24 ment in the scoping unit of a module. Note that this prohibits the local name from appearing in  
25 COMMON and EQUIVALENCE specifications, but permits the appearance of local names in  
26 namelist group lists. The appearance of such a local name in a PUBLIC statement in a module  
27 causes the entity accessible by the USE statement to be a public entity of that module. If the name  
28 appears in a PRIVATE statement in a module, the entity is not a public entity of that module. If  
29 the local name does not appear in either a PUBLIC or PRIVATE statement, it assumes the default  
30 accessibility attribute (5.2.3) of that scoping unit. no  
change
- 31 Examples:
- 32 USE STATS\_LIB
- 33 provides access to all public entities in the module STATS\_LIB.
- 34 USE MATH\_LIB; USE STATS\_LIB, SPROD => PROD
- 35 makes all public entities in both MATH\_LIB and STATS\_LIB accessible. If MATH\_LIB contains an  
36 entity called PROD, it is accessible by its own name while the entity PROD of STATS\_LIB is acces-  
37 sible by the name SPROD.
- 38 USE STATS\_LIB, ONLY: YPROD; USE STATS\_LIB, ONLY : PROD
- 39 makes public entities YPROD and PROD in STATS\_LIB accessible.
- 40 USE STATS\_LIB, ONLY : YPROD; USE STATS\_LIB
- 41 makes all public entities in STATS\_LIB accessible.
- 42 **11.3.3 Examples of the Use of Modules.**
- 43 **11.3.3.1 Identical Common Blocks.** A common block and all its associated specification state-  
44 ments may be placed in a module named, for example, MY\_COMMON and accessed by a USE  
45 statement of the form
- 46 USE MY\_COMMON

## Program Units

1 that accesses the whole module without any renaming. This ensures that all instances of the com-  
2 mon block are identical. Module MY\_COMMON could contain more than one common block.

3 **11.3.3.2 Global Data.** A module may contain only data objects, for example:

```
4 MODULE DATA_MODULE
5     SAVE
6     REAL A (10), B, C (20,20)
7     INTEGER :: I=0
8     INTEGER, PARAMETER :: J=10
9     COMPLEX D (J,J)
10  END MODULE
```

11 Note that data objects made global in this manner may have any combination of data types.

12 Access to some of these may be made by a USE statement with the ONLY option, such as:

```
13 USE DATA_MODULE, ONLY: A, B, D
```

14 and access to all of them may be made by the following USE statement:

```
15 USE DATA_MODULE
```

16 Access to all of them with some renaming to avoid name conflicts may be made by:

```
17 USE DATA_MODULE, AMODULE => A, DMODULE => D
```

18 **11.3.3.3 <sup>Derived Types</sup> Data Structures.** A derived type may be defined in a module and accessed in a number  
19 of program units. For example:

```
20 MODULE SPARSE
21     TYPE NONZERO
22         REAL A
23         INTEGER I, J
24     END TYPE
25 END MODULE
```

26 defines a type consisting of a real component and two integer components for holding the numeri-  
27 cal value of a nonzero matrix element and its row and column indices.

28 **11.3.3.4 Global Allocatable Arrays.** Many programs need large global allocatable arrays whose  
29 sizes are not known before program execution. A simple form for such a program is:

```
30 PROGRAM GLOBAL_WORK
31     CALL CONFIGURE_ARRAYS      ! Perform the appropriate allocations
32     CALL COMPUTE               ! Use the arrays in computations
33 END PROGRAM GLOBAL_WORK

34 MODULE WORK_ARRAYS           ! An example set of work arrays
35     INTEGER N
36     REAL, ALLOCATABLE, SAVE :: A (:), B (:, :), C (:, :, :)
37 END MODULE WORK_ARRAYS

38 SUBROUTINE CONFIGURE_ARRAYS  ! Process to set up work arrays
39     USE WORK_ARRAYS
40     READ (*, *) N
41     ALLOCATE (A (N), B (N, N), C (N, N, 2 * N))
42 END SUBROUTINE CONFIGURE_ARRAYS
```



```

1  SUBROUTINE COMPUTE
2      USE WORK_ARRAYS
3      ... ! Computations involving arrays A, B, and C
4  END SUBROUTINE COMPUTE

```

5 Typically, many subprograms need access to the work arrays, and all such subprograms would  
6 contain the statement

```
7  USE WORK_ARRAYS
```

8 **11.3.3.5 Procedure Libraries.** Interface blocks for external procedures in a library may be gath-  
9 ered into a module. This permits the use of argument keywords and optional arguments, and  
10 allows static checking of the references. Different versions may be constructed for different appli-  
11 cations, using keywords in common use in each application. An example is the following library  
12 module:

```

13  MODULE LIBRARY_LLS
14      INTERFACE
15          SUBROUTINE LLS (X, A, F, FLAG)
16              REAL X (:, :)
17              ! The SIZE in the next statement is an intrinsic function
18              REAL, DIMENSION (SIZE (X, 2)) :: A, F
19              INTEGER FLAG
20          END SUBROUTINE LLS
21          ...
22      END INTERFACE
23      ...
24  END MODULE LIBRARY_LLS

```

25 This module allows the subroutine LLS to be invoked:

```

26  USE LIBRARY_LLS
27      ...
28  CALL LLS (X = ABC, A = D, F = XX, FLAG = IFLAG)
29      ...

```

30 **11.3.3.6 Operator Extensions.** In order to extend an intrinsic operator symbol to have an addi-  
31 tional meaning, an interface block specifying that operator symbol in the OPERATOR option of the  
32 INTERFACE statement may be placed in a module. For example, // may be extended to perform  
33 concatenation of two derived-type objects serving as varying length character strings and + may  
34 be extended to specify matrix addition for type MATRIX or interval arithmetic addition for type  
35 INTERVAL.

36 A module might contain several such interface blocks. An operator may be defined by an external  
37 function (either in Fortran or some other language) and its procedure interface placed in the mod-  
38 ule.

39 **11.3.3.7 Data Abstraction.** A module may encapsulate a derived-type definition and all the pro-  
40 cedures that represent operations on values of this type. An example is given in C.11.5 for set  
41 operations.

42 **11.3.3.8 Public Entities Renamed.** At times it may be necessary to rename entities that are  
43 accessed with USE statements. Care must be taken if the modules contain USE statements, too.

44 The following example illustrates renaming features of the USE statement. <sup>reference also</sup>

## Program Units

```
1  MODULE J; REAL JX, JY, JZ; END MODULE
2  MODULE K
3      USE J, ONLY : KX => JX, KY => JY
4      ! KX and KY are local names to module K
5      REAL KZ          ! KZ is local name to module K
6      REAL JZ          ! JZ is local name to module K
7  END MODULE

8  PROGRAM RENAME
9      USE J; USE K
10     ! Module J's entity JX is accessible under names JX and KX
11     ! Module J's entity JY is accessible under names JY and KY
12     ! Module K's entity KZ is accessible under name KZ
13     ! Module J's entity JZ and K's entity JZ are different entities
14     ! and must not be referenced
15     ...
16 END PROGRAM RENAME
```

17 **11.4 Block Data Program Units.** A block data program unit is used to provide initial values  
18 for data objects in named common blocks.

```
19 R1110 block-data                is block-data-stmt
20                                     [ specification-part ]
21                                     end-block-data-stmt

22 R1111 block-data-stmt            is BLOCK DATA [ block-data-name ]

23 R1112 end-block-data-stmt        is END [ BLOCK DATA [ block-data-name ] ]
```

24 **Constraint:** The *block-data-name* may be included in the *end-block-data-stmt* only if it was provided  
25 in the *block-data-stmt* and, if included, must be identical to the *block-data-name* in the  
26 *block-data-stmt*.

27 **Constraint:** A *block-data specification-part* may contain only USE statements, type declaration statements,  
28 IMPLICIT statements, PARAMETER statements, derived-type definitions,  
29 and the following specification statements: COMMON, DATA, DIMENSION,  
30 EQUIVALENCE, INTRINSIC, POINTER, SAVE, and TARGET.

31 **Constraint:** A type declaration statement in a *block-data specification-part* must not contain ALLO-  
32 CATABLE, EXTERNAL, INTENT, OPTIONAL, PRIVATE, or PUBLIC attribute spec-  
33 ifiers.

34 If an object in a named common block is initially defined, all objects having storage units in the  
35 common block storage sequence must be specified even if they are not all initially defined. More  
36 than one named common block may have objects initially defined in a single block data program  
37 unit.

38 Only a nonpointer object in a named common block may be initially defined in a block data pro-  
39 gram unit. Note that objects associated with an object in a common block are considered to be in  
40 that common block.

41 The same named common block must not be specified in more than one block data program unit  
42 in an executable program.

43 There must not be more than one unnamed block data program unit in an executable program.

44 An example of a block data program unit is:

```
45 BLOCK DATA WORK
46     COMMON /WRKCOM/ A, B, C (10, 10)
47     DATA A /1.0/, B /2.0/, C /100 * 0.0/
48 END BLOCK DATA WORK
```

## 12. PROCEDURES

1 The concept of a procedure was introduced in 2.2.3. This section contains a complete description  
2 of procedures. The actions specified by a procedure are performed when the procedure is invoked  
3 by execution of a reference to it. The reference may identify, as actual arguments, entities that are  
4 associated during execution of the procedure reference with corresponding dummy arguments in  
5 the procedure definition.

6 **12.1 Procedure Classifications.** A procedure is classified according to the form of its refer-  
7 ence and the way it is defined.

8 **12.1.1 Procedure Classification by Reference.** The definition of a procedure specifies it to be a  
9 function or a subroutine. A reference to a function either appears explicitly as a primary within an  
10 expression, or is implied by a defined operation within an expression. A reference to a subroutine  
11 is a CALL statement or a defined assignment statement (7.5.1.3).

12 A procedure is classified as **elemental** if it is an intrinsic procedure that may be referenced elemen-  
13 tally (12.4.3, 12.4.5).

14 **12.1.2 Procedure Classification by Means of Definition.** A procedure is either an intrinsic pro-  
15 cedure, an external procedure, a module procedure, an internal procedure, a dummy procedure,  
16 or a statement function.

17 **12.1.2.1 Intrinsic Procedures.** A procedure that is provided as an inherent part of the processor  
18 is an **intrinsic procedure**.

19 **12.1.2.2 External, Internal, and Module Procedures.** An external procedure is a procedure  
20 that is defined by an external subprogram or by a means other than Fortran.

21 An **internal procedure** is a procedure that is defined by an internal subprogram. Internal proce-  
22 dures may appear in the main program, in an external subprogram, or in a module subprogram.  
23 Internal procedures must not appear in other internal procedures. Internal procedures are the  
24 same as external procedures except that the name of the internal procedure is not a global entity,  
25 an internal procedure must not contain an ENTRY statement, the internal procedure name must  
26 not be argument associated with a dummy procedure (12.4.1.2), and the internal procedure has  
27 access to host entities by host association.

28 A **module procedure** is a procedure that is defined by a module subprogram.

29 If a subprogram contains one or more ENTRY statements, it defines a procedure for each ENTRY  
30 statement and a procedure for the SUBROUTINE or FUNCTION statement.

31 **12.1.2.2.1 Host Association.** An internal subprogram, a module subprogram, or a derived-type  
32 definition has access to the named entities from its host via **host association**. The accessed entities  
33 are known by the same name and have the same attributes as in the host and are variables, con-  
34 stants, procedures including interfaces, derived types, type parameters, derived-type components,  
35 and namelist groups.

36 Any declaration or specification of an entity, or any entity accessed through use association, with  
37 the same ~~non~~generic name as an accessible entity from the host makes the host entity inaccessible.  
38 The appearance of a name as a dummy argument or as a function result is treated as a specifica-  
39 tion. Entities that are local to a procedure are not accessible to its host.

40 Note that an interface body does not access the named entities by host association, but it may  
41 access entities by use association (11.3.2).

42 If a procedure gains access to a pointer by host association, the association of the pointer with a  
43 target that is current at the time the procedure is invoked remains current within the procedure.  
44 This pointer association may be changed within the procedure. When execution of the procedure  
45 completes, the pointer association that was current remains current, except where the associated  
46 target is a dummy argument or was declared within the procedure and is not saved. In these  
47 cases, the completion of the procedure causes the pointer association status of the host associated

*Completion causes the target to become undefined (Item 4) of 14.2.6.*

*S/B  
in ch  
14?*

*? not  
in  
14.2.6*

## Procedures

1 pointer to become undefined.

2 **12.1.2.2 Host Association and USE Association.** A host procedure and an internal procedure  
3 may contain the same and differing use-associated entities, as illustrated in the following example.

```
4 MODULE B; REAL BX, Q; INTEGER IX, JX; END MODULE
5 MODULE C; REAL CX; END MODULE
6 MODULE D; REAL DX, DY, DZ; END MODULE
7 MODULE E; REAL EX, EY, EZ; END MODULE
8 MODULE F; REAL FX; END MODULE
9 MODULE G; USE F; REAL GX; END MODULE

10 PROGRAM A
11 USE B; USE C; USE D
12 ...
13 CONTAINS
14     SUBROUTINE INNER_PROC (Q)
15         USE C           ! Not needed
16         USE B, ONLY: BX ! Entities accessible are BX, IX, and JX
17                         ! if no other IX or JX
18                         ! is accessible to INNER_PROC
19                         ! Q is local to INNER_PROC
20         USE D, X => DX  ! Entities accessible are DX, DY, and DZ
21                         ! X is local name for DX in INNER_PROC
22                         ! X and DX denote same entity if no other
23                         ! entity DX is local to INNER_PROC
24         USE E, ONLY: EX ! EX is accessible in INNER_PROC, not in program A
25                         ! EY and EZ are not accessible in INNER_PROC
26                         ! or in program A
27         USE G           ! FX and GX are accessible in INNER_PROC
28         ...
29     END SUBROUTINE INNER_PROC
30 END PROGRAM A
```

31 Note: Because program A contains the statement

32 USE B

33 all of the entities in module B, except for Q, are accessible in INNER\_PROC, even though  
34 INNER\_PROC contains the statement

35 USE B, ONLY: BX

36 The USE statement with the ONLY keyword means that this particular statement brings in only  
37 the entity named, not that this is the only variable from the module accessible in this scoping unit.

38 **12.1.2.3 Dummy Procedures.** A dummy argument that is specified as a procedure or appears in  
39 a procedure reference is a dummy procedure.

40 **12.1.2.4 Statement Functions.** A function that is defined by a single statement is a statement  
41 function (12.5.4).

42 **12.2 Characteristics of Procedures.** The characteristics of a procedure are the classification  
43 of the procedure as a function or subroutine, the characteristics of its arguments, and the character-  
44 istics of its result value if it is a function.

45 **12.2.1 Characteristics of Dummy Arguments.** Each dummy argument is either a dummy data  
46 object, a dummy procedure, or an asterisk (alternate return indicator). A dummy argument other than an aster-  
47 isk may be specified to have the OPTIONAL attribute. This attribute means that the dummy argu-  
48 ment need not be associated with an actual argument for any particular reference to the procedure.

1 **12.2.1.1 Characteristics of Dummy Data Objects.** The characteristics of a dummy data object  
 2 are its type, its type parameters (if any), its shape, its intent (5.1.2.3, 5.2.1), whether it is optional  
 3 (5.1.2.6, 5.2.2), and whether it is a pointer (5.1.2.7, 5.2.7) or a target (5.1.2.8, 5.2.8). If a type param-  
 4 eter of an object or a bound of an array is an expression that depends on the value or attributes of  
 5 another object, the exact dependence on other entities is a characteristic. If a shape, size, or charac-  
 6 ter length is assumed, it is a characteristic.

7 **12.2.1.2 Characteristics of Dummy Procedures.** The characteristics of a dummy procedure are  
 8 the explicitness of its interface (12.3.1), its characteristics as a procedure if the interface is explicit,  
 9 and whether it is optional (5.1.2.6, 5.2.2).

10 **12.2.1.3 Characteristics of Asterisk Dummy Arguments.** An asterisk as a dummy argument has no char-  
 11 aracteristics.

12 **12.2.2 Characteristics of Function Results.** The characteristics of a function result are its type,  
 13 type parameters (if any), rank, and whether it is a pointer. If a function result is an array that is not  
 14 a pointer, its shape is a characteristic. Where a type parameter or bound of an array is not a con-  
 15 stant expression, the exact dependence on the entities in the expression is a characteristic. If the  
 16 length of a character data object is assumed, this is a characteristic.

17 **12.3 Procedure Interface.** The interface of a procedure determines the forms of reference  
 18 through which it may be invoked. The interface consists of the characteristics of the procedure, the  
 19 name of the procedure, the name and characteristics of each dummy argument, and the  
 20 procedure's generic identifiers, if any. The characteristics of a procedure are fixed, but the remain-  
 21 der of the interface may differ in different scoping units.

22 **12.3.1 Implicit and Explicit Interfaces.** If a procedure is accessible in a scoping unit, its interface  
 23 is either **explicit** or **implicit** in that scoping unit. The interface of an internal procedure, module  
 24 procedure, or intrinsic procedure is always explicit in such a scoping unit. The interface of a recur-  
 25 sive subroutine or a recursive function with a separate result name is explicit within the subpro-  
 26 gram that defines it. The interface of a statement function is always implicit. The interface of an  
 27 external procedure or dummy procedure is explicit in a scoping unit other than its own if an inter-  
 28 face block (12.3.2.1) for the procedure is supplied or accessible, and implicit otherwise. For exam-  
 29 ple, the subroutine LLS of 11.3.3.5 has an explicit interface.

30 **12.3.1.1 Explicit Interface.** A procedure must have an explicit interface if any of the following is  
 31 true:

- 32 (1) A reference to the procedure appears:
- 33 (a) With an argument keyword (12.4.1)
- 34 (b) As a defined assignment (subroutines only)
- 35 (c) In an expression as a defined operator (functions only)
- 36 (d) As a reference by its generic name (12.3.2.1)
- 37 (2) The procedure has:
- 38 (a) An optional dummy argument
- 39 (b) An array-valued result (functions only)
- 40 (c) A dummy argument that is an assumed-shape array, a pointer, or a target
- 41 (d) A result whose length type parameter value is neither assumed nor constant (char-
- 42 acter functions only)
- 43 (e) A result that is a pointer (functions only)

## Procedures

1 **12.3.2 Specification of the Procedure Interface.** The interface for an internal, external, mod-  
2 ule, or dummy procedure is specified by a FUNCTION, SUBROUTINE, or ENTRY statement and  
3 by specification statements for the dummy arguments and the result of a function. These state-  
4 ments may appear in the procedure definition, in an interface block, or in both except that the  
5 ENTRY statement must not appear in an interface block. Note that internal procedures must not  
6 appear in an interface block.

### 7 12.3.2.1 Procedure Interface Block.

8 R1201 *interface-block* is *interface-stmt*  
9 [ *interface-body* ] ...  
10 [ *module-procedure-stmt* ] ...  
11 *end-interface-stmt*

12 R1202 *interface-stmt* is INTERFACE [ *generic-spec* ]

13 R1203 *end-interface-stmt* is END INTERFACE

14 R1204 *interface-body* is *function-stmt*  
15 [ *specification-part* ]  
16 *end-function-stmt*  
17 or *subroutine-stmt*  
18 [ *specification-part* ]  
19 *end-subroutine-stmt*

20 R1205 *module-procedure-stmt* is MODULE PROCEDURE *procedure-name-list*

21 R1206 *generic-spec* is *generic-name*  
22 or OPERATOR ( *defined-operator* )  
23 or ASSIGNMENT ( = )

24 Constraint: An *interface-body* must not contain an *entry-stmt*, *data-stmt*, *format-stmt*, or *stmt-*  
25 *function-stmt*.

26 Constraint: The MODULE PROCEDURE specification is allowed only if the *interface-block* has a  
27 *generic-spec*, and has a host that is a module or accesses a module by  
28 association; each procedure name must be the name of a module procedure  
29 that is accessible in the host.

28 Constraint: An *interface-block* must not appear in a BLOCK DATA program unit.

29 Constraint: An *interface-block* in a subprogram must not contain an *interface-body* for a procedure  
30 defined by that subprogram.

31 An external or module subprogram definition specifies a specific interface for the procedures  
32 defined in that subprogram. Such a specific interface is explicit for module procedures and  
33 implicit for external procedures. An interface body in an interface block specifies an explicit inter-  
34 face for an existing external procedure or a dummy procedure. If the name on a procedure head-  
35 ing in an interface block is the same as the name of a dummy argument in the subprogram contain-  
36 ing the interface block, the interface block declares that dummy argument to be a dummy proced-  
37 ure with the indicated interface; otherwise, the interface block declares the name to be the name  
38 of an external procedure with the indicated procedure interface.

39 An interface body specifies all of the procedure's characteristics and these must be consistent with  
40 those specified in the procedure definition. Note that the dummy argument names may be differ-  
41 ent. An interface block must not contain an ENTRY statement, but an ENTRY interface may be  
42 specified by using the entry name as the procedure name in the interface body. A procedure must  
43 not have more than one explicit specific interface in a given scoping unit.

44 An example of an interface block without a generic specification is:

45 INTERFACE

46 SUBROUTINE EXT1 (X, Y, Z)

47 REAL, DIMENSION (100, 100) :: X, Y, Z

48 END SUBROUTINE EXT1

conflicts  
w/ 12.1.2.2.1

```

1  SUBROUTINE EXT2 (X, Z)  REAL X
2    COMPLEX (KIND = 4) Z (2000)
3  END SUBROUTINE EXT2

4  FUNCTION EXT3 (P, Q)
5    LOGICAL EXT3
6    INTEGER P (1000)
7    LOGICAL Q (1000)
8  END FUNCTION EXT3

```

```
9  END INTERFACE
```

10 This interface block specifies explicit interfaces for the three external procedures EXT1, EXT2, and  
11 EXT3. Any of these procedures may use keyword calls; for example:

```
12 EXT3 (Q = P_MASK (N+1 : N+1000), P = ACTUAL_P)
```

13 An interface block with a generic specification specifies a **generic interface** for each of the proce-  
14 dures in the interface block. If the host is a module or accesses a module by use association, the  
15 MODULE PROCEDURE specification lists those module procedures, either defined in that module  
16 or accessible via a USE statement, that have this generic interface. The characteristics of module  
17 procedures are not given in interface blocks, but are assumed from the module subprogram defini-  
18 tions. A generic interface is always explicit.

19 A procedure always may be referenced via its specific interface. It also may be referenced via its  
20 generic interface, if it has one. The generic name, defined operator, or equals symbol in a generic  
21 specification is a **generic identifier** for all the procedures in the interface block. The rules on how  
22 any two procedures with the same generic identifier must differ are given in 14.1.2.3. They ensure  
23 that any generic invocation applies to at most one specific procedure.

24 A **generic name** specifies a single name to reference all of the procedure names in the interface  
25 block. A generic name may be the same as any one of the procedure names in ~~that~~ <sup>the</sup> interface block,  
26 or the same as any accessible generic name.

27 An example of a generic procedure interface is:

```

28  INTERFACE SWITCH

29    SUBROUTINE INT_SWITCH (X, Y)
30      INTEGER, INTENT (INOUT) :: X, Y
31    END SUBROUTINE INT_SWITCH

32    SUBROUTINE REAL_SWITCH (X, Y)
33      REAL, INTENT (INOUT) :: X, Y
34    END SUBROUTINE REAL_SWITCH

35    SUBROUTINE COMPLEX_SWITCH (X, Y)
36      COMPLEX, INTENT (INOUT) :: X, Y
37    END SUBROUTINE COMPLEX_SWITCH

```

```
38  END INTERFACE
```

39 Any of these three subroutines (INT\_SWITCH, REAL\_SWITCH, COMPLEX\_SWITCH) may be ref-  
40 erenced with the generic name SWITCH, as well as by its ~~original~~ <sup>specific</sup> name. For example, a reference  
41 to INT\_SWITCH could take the form:

```
42 CALL SWITCH (MAX_VAL, LOC_VAL) ! MAX_VAL and LOC_VAL are of type INTEGER
```

43 **12.3.2.1.1 Defined Operations.** If OPERATOR is specified in a generic specification, all of the  
44 procedures specified in the interface block must be functions, that may be referenced as defined  
45 operations (12.4). In the case of functions of two arguments, infix binary operator notation is  
46 implied. In the case of functions of one argument, prefix operator notation is implied.

## Procedures

1 OPERATOR must not be specified for functions with no arguments or for functions with more  
2 than two arguments. If the operator is an *intrinsic-operator* (R310), the number of function argu-  
3 ments must be consistent with the intrinsic uses of that operator. In addition, the dummy argu-  
4 ments must be nonoptional dummy data objects and must be specified with INTENT (IN), and the  
5 *function result must not have assumed character length.*  
6 A defined operation is treated as a reference to the function. For a unary defined operation, the  
7 operand corresponds to the function's dummy argument; for a binary operation, the left-hand  
8 operand corresponds to the first dummy argument of the function and the right-hand operand  
9 corresponds to the second argument.

9 An example of the use of the OPERATOR generic specification is:

```
10 INTERFACE OPERATOR ( * )
11     FUNCTION BOOLEAN_AND (B1, B2)
12         LOGICAL, INTENT (IN) :: B1 (:), B2 (SIZE (B1))
13         LOGICAL :: BOOLEAN_AND (SIZE (B1))
14     END FUNCTION BOOLEAN_AND
15 END INTERFACE
```

16 This allows, for example

```
17 SENSOR (1:N) * ACTION (1:N)
```

18 as an alternative to the function call

```
19 BOOLEAN_AND (SENSOR (1:N), ACTION (1:N)) ! SENSOR and ACTION are
20   ! of type LOGICAL
```

21 A given defined operator may, as with generic names, apply to more than one function, in which  
22 case it is generic in exact analogy to generic procedure names. For intrinsic operator symbols, the  
23 generic properties include the intrinsic operations they represent. Because both forms of each rela-  
24 tional operator have the same interpretation (7.3), overloading one form (such as <=) has the effect  
25 of defining both forms (<= and .LE.).

*Each argument must be nonoptional.*

26 **12.3.2.1.2 Defined Assignments.** If ASSIGNMENT is specified in an INTERFACE statement, all  
27 the procedures in the interface block must be subroutines that may be referenced as defined  
28 assignments (7.5.1.3). Each of these subroutines must have exactly two dummy arguments. The  
29 first argument must have INTENT (OUT) or INTENT (INOUT) and the second argument must  
30 have INTENT (IN). A defined assignment is treated as a reference to the subroutine, with the left-  
31 hand side as the first argument and the right-hand side enclosed in parentheses as the second  
32 argument. The ASSIGNMENT generic specification specifies that the assignment operation is  
33 extended or redefined if both sides of the equals sign are of the same derived type.

34 An example of the use of the ASSIGNMENT generic specification is

```
35 INTERFACE ASSIGNMENT ( = )
36     SUBROUTINE BIT_TO_NUMERIC (N, B)
37         INTEGER, INTENT (OUT) :: N
38         LOGICAL, INTENT (IN)  :: B (:)
39     END SUBROUTINE BIT_TO_NUMERIC
40     SUBROUTINE CHAR_TO_STRING (S, C)
41         USE STRING_MODULE ! contains definition of type STRING
42         TYPE (STRING), INTENT (OUT) :: S ! A variable-length string
43         CHARACTER (*), INTENT (IN)  :: C
44     END SUBROUTINE CHAR_TO_STRING
45 END INTERFACE
```



1 Example assignments are:

```
2 KOUNT = SENSOR (J:K) ! CALL BIT_TO_NUMERIC (KOUNT, (SENSOR (J:K)))
3 NOTE = '89AB' ! CALL CHAR_TO_STRING (NOTE, ('89AB'))
```

4 **12.3.2.2 EXTERNAL Statement.** An EXTERNAL statement is used to specify a name as representing an external procedure, a dummy procedure, or a block data program unit. Specifying an external procedure name or a dummy procedure name in an EXTERNAL statement permits such a name to be used as an actual argument.

8 R1207 *external-stmt* is EXTERNAL *external-name-list*

9 Each *external-name* must be the name of an external procedure, a dummy argument, or a block data program unit.

11 The appearance of the name of a dummy argument in an EXTERNAL statement specifies that the dummy argument is a dummy procedure.

13 The appearance in an EXTERNAL statement of a name that is not the name of a dummy argument specifies that the name is the name of an external procedure or block data program unit. If an external procedure name or a dummy procedure name is used as an actual argument, it must appear in an EXTERNAL statement, be given the external attribute in a type declaration statement, or be declared to be a procedure by an interface block in the scoping 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.

21 Only one appearance of a name in all of the EXTERNAL statements in a scoping unit is permitted.

22 An example of an EXTERNAL statement is:

```
23 SUBROUTINE SUB (FOCUS)
24 EXTERNAL FOCUS
```

25 **12.3.2.3 INTRINSIC Statement.** An INTRINSIC statement specifies a list of names that have the INTRINSIC attribute. A name that has the INTRINSIC attribute represents an intrinsic procedure (Section 13). The INTRINSIC attribute permits a name that represents a specific intrinsic function to be used as an actual argument.

29 R1208 *intrinsic-stmt* is INTRINSIC *intrinsic-procedure-name-list*

30 Constraint: Each *intrinsic-procedure-name* must be the name of an intrinsic procedure.

31 The appearance of a name in an INTRINSIC statement confirms that the name is the name of an intrinsic procedure. The appearance of a generic intrinsic function name (13.10) in an INTRINSIC statement does not cause that name to lose its generic property.

34 If the specific name (13.12) of an intrinsic function is used as an actual argument, the name must either appear in an INTRINSIC statement or be given the intrinsic attribute in a type declaration statement in the scoping unit.

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

40 **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 the procedure is implicit must be such that the actual arguments are consistent with the characteristics of the dummy arguments.

## Procedures

1 **12.4 Procedure Reference.** The form of a procedure reference is dependent on the interface  
2 of the procedure, but is independent of the means by which the procedure is defined. The forms of  
3 procedure references are:

4 R1209 *function-reference* is *function-name* ( [ *actual-arg-spec-list* ] )

5 Constraint: The *actual-arg-spec-list* for a function reference must not contain an *alt-return-spec*.

6 R1210 *call-stmt* is CALL *subroutine-name* [ ( [ *actual-arg-spec-list* ] ) ]

7 A function may be referenced also as a defined operation and a subroutine may be referenced also  
8 as a defined assignment.

### 9 12.4.1 Actual Argument List.

10 R1211 *actual-arg-spec* is [ *keyword* = ] *actual-arg*

11 R1212 *keyword* is *dummy-arg-name*

12 R1213 *actual-arg* is *expr*  
13 or *variable*  
14 or *procedure-name*  
15 or *alt-return-spec*

16 R1214 *alt-return-spec* is \* *label*

17 Constraint: The *keyword* = must not appear if the interface of the procedure is implicit in the  
18 scoping unit.

19 Constraint: The *keyword* = may be omitted from an *actual-arg-spec* only if the *keyword* = has been  
20 omitted from each preceding *actual-arg-spec* in the argument list.

21 Constraint: Each *keyword* must be the name of a dummy argument in the explicit interface of the  
22 procedure.

23 Constraint: A *procedure-name actual-arg* must not be the name of an internal procedure and must  
24 not be the generic name of a procedure (12.3.2.1, 13.1).

25 Constraint: The *label* used in the *alt-return-spec* must be the statement label of a branch target statement that appears in  
26 the same scoping unit as the *call-stmt*.

27 In either a subroutine reference or a function reference, the actual argument list identifies the cor-  
28 respondence between the actual arguments supplied and the dummy arguments of the procedure.  
29 In the absence of a keyword, an actual argument is associated with the dummy argument occupy-  
30 ing the corresponding position in the dummy argument list; that is, the first actual argument is  
31 associated with the first dummy argument, the second actual argument is associated with the sec-  
32 ond dummy argument, etc. If a keyword is present, the actual argument is associated with the  
33 dummy argument whose name is the same as the keyword (using the dummy argument names  
34 from the interface accessible in the scoping unit containing the procedure reference). Exactly one  
35 actual argument must be associated with each nonoptional dummy argument. At most one actual  
36 argument may be associated with each optional dummy argument. Each actual argument must be  
37 associated with a dummy argument. For example, the procedure

```
38 SUBROUTINE SOLVE (FUNCT, SOLUTION, METHOD, STRATEGY, PRINT)
39   INTERFACE
40     FUNCTION FUNCT (X)
41     REAL FUNCT, X
42   END FUNCTION FUNCT
43 END INTERFACE
44 REAL SOLUTION
45 INTEGER, OPTIONAL :: METHOD, STRATEGY, PRINT
46 ...
```

47 may be invoked with

The character length of an actual argument of type nondefault character must agree with that of the dummy argument. Procedures

1 CALL SOLVE (FUN, SOL, PRINT = 6)  
2 providing its interface is explicit.

3 **12.4.1.1 Arguments Associated with Dummy Data Objects.** If a dummy argument is a  
4 dummy data object, the associated actual argument must be an expression of the same type or a  
5 data object of the same type. The kind type parameter value of the actual argument must agree  
6 with that of the dummy argument. If a scalar dummy argument is of type character, the length *len*  
7 of the dummy argument must be less than or equal to the length of the actual argument. The  
8 dummy argument becomes associated with the leftmost *len* characters of the actual argument. If  
9 an array dummy argument is of type character, the restriction on length is for the entire array and  
10 not for each array element. The length of an array element in the dummy argument array may be  
11 different from the length of an array element in the associated actual argument array, array ele-  
12 ment, or array element substring, but the dummy argument array must not extend beyond the end  
13 of the actual argument array. Except when a procedure reference is elemental (12.4.3, 12.4.5), each  
14 element of an array-valued actual argument or of a sequence in a sequence association (12.4.1.4) is  
15 associated with the element of the dummy array that has the same position in array element order  
16 (6.2.2.2).

17 If the dummy argument is a pointer, the actual argument must be a pointer and the types, type  
18 parameters, and ranks must agree.

19 At the invocation of the procedure, the dummy argument pointer receives the pointer association  
20 status of the actual argument. If the actual argument is currently associated, the dummy argument  
21 becomes associated with the same target. The association status may change during the execution  
22 of the procedure. When execution of the procedure completes, the pointer association status of the  
23 dummy argument becomes undefined if it is associated with a dummy argument of the procedure  
24 that has the TARGET attribute or with a target that becomes undefined (14.7.6); following this, the  
25 pointer association status of the actual argument becomes that of the dummy argument.

26 If the actual argument has the TARGET attribute, any pointers associated with it do not become  
27 associated with the corresponding dummy argument on invocation of the procedure, but remain  
28 associated with the actual argument. If the dummy argument has the TARGET attribute, any  
29 pointer associated with it becomes undefined when execution of the procedure completes.

30 If the actual argument is scalar, the corresponding dummy argument must be scalar unless the  
31 actual argument is an element of a named array that is explicit-shaped or assumed-size, or a sub-  
32 string of such an element. If the procedure is referenced by a generic name or as a defined opera-  
33 tor or defined assignment, the ranks of the actual arguments and corresponding dummy argu-  
34 ments must agree.

35 If a dummy argument has INTENT (OUT) or INTENT (INOUT), the actual argument must be  
36 definable. If a dummy argument has INTENT (OUT), the corresponding actual argument becomes  
37 undefined at the time the association is established.

38 If the actual argument is an array section having a vector subscript, the dummy argument is not  
39 definable and must not have INTENT (OUT) or INTENT (INOUT).

40 If a dummy argument is an assumed-shape array, the actual argument must not be an assumed-  
41 size array or a scalar (including an array element designator or an array element substring desig-  
42 nator).

43 A scalar dummy argument may be associated only with a scalar actual argument.

44 **12.4.1.2 Arguments Associated with Dummy Procedures.** If a dummy argument is a dummy  
45 procedure, the associated actual argument must be the specific name of an external, module,  
46 dummy, or intrinsic procedure. The only intrinsic procedures permitted are those listed in 13.12  
47 and not marked with a bullet (•). If the specific name is also a generic name, only the specific pro-  
48 cedure is associated with the dummy argument.

49 If the interface of the dummy procedure is explicit, the characteristics of the associated actual pro-  
50 cedure must be the same as the characteristics of the dummy procedure (12.2).

actual on 112 ?

, if any, 116

no discussion there

not an assumed shape or pointer array

## Procedures

- 1 If the interface of the dummy procedure is implicit and either the name of the dummy procedure is
- 2 explicitly typed or the procedure is referenced as a function, the dummy procedure must not be
- 3 referenced as a subroutine and the actual argument must be a function or dummy procedure.
- 4 If the interface of the dummy procedure is implicit and a reference to the procedure appears as a
- 5 subroutine reference, the actual argument must be a subroutine or dummy procedure.

6 **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

7 must identify an executable construct in the scoping unit containing the procedure reference.

8

9 **12.4.1.4 Sequence Association.** An actual argument represents an element sequence if it is an

10 array expression, an array element designator, or an array element substring designator. If the

11 actual argument is an array expression, the element sequence consists of the elements in array element

12 order. If the actual argument is an ~~array~~ array element designator, the element sequence

13 consists of that array element and each element that follows it in array element order.

14 If the actual argument is <sup>of type default character and is an</sup> ~~a~~ character array expression, ~~character~~ array element, or ~~character~~ array

15 element substring designator, the element sequence consists of the character storage units begin-

16 ning with the first storage unit of the actual argument and continuing to the end of the array. The

17 character storage units of an array element substring designator are viewed as array elements con-

18 sisting of consecutive groups of character storage units having the ~~length of the array element sub-~~

19 ~~string.~~ Thus, the first such element is the array element ~~substring itself.~~

20 elements in the element sequence may consist of storage units from different elements of the original

21 array. *character length of the dummy array.*

22 An actual argument that represents an element sequence and corresponds to a dummy argument

23 that is an array-valued data object is sequence associated with the dummy argument if the dummy

24 argument is an explicit-shape or assumed-size array. The rank and shape of the actual argument

25 need not agree with the rank and shape of the dummy argument, but the number of elements in

26 the dummy argument must not exceed the number of elements in the element sequence of the

27 actual argument. If the dummy argument is assumed size, the number of elements in the dummy

28 argument is exactly the number of elements in the element sequence.

29 **12.4.2 Function Reference.** A function is invoked during expression evaluation by a function

30 reference or by a defined operation (7.1.3). When it is invoked, all actual argument expressions are

31 evaluated, then the arguments are associated, and then the function is executed. When execution

32 of the function is complete, the value of the function result is available for use in the expression

33 that caused the function to be invoked. The characteristics of the function result (12.2.2) are deter-

34 mined by the interface of the function.

35 **12.4.3 Elemental Intrinsic Function Reference.** A reference to an elemental intrinsic function is

36 an elemental reference if one or more actual arguments are arrays and all array arguments have

37 the same shape. The result has the same shape as the array arguments and the value of each ele-

38 ment in the result is obtained by evaluating the function using the scalar arguments and the corre-

39 sponding elements of the array arguments. For example, if X and Y are arrays of shape (m, n),

40  $\text{MAX}(X, 0.0, Y)$

41 is an array expression of shape (m, n) whose elements have values

42  $\text{MAX}(X(i, j), 0.0, Y(i, j)), i = 1, 2, \dots, m, j = 1, 2, \dots, n$

43 **12.4.4 Subroutine Reference.** A subroutine is invoked by execution of a CALL statement or

44 defined assignment statement (7.5.1.3). When a subroutine is invoked, all actual argument expres-

45 sions are evaluated, then the arguments are associated, and then the subroutine is executed. When

46 the actions specified by the subroutine are completed, execution of the CALL statement or defined

47 assignment statement is also completed. If a CALL statement includes one or more alternate return specifiers

48 among its arguments, control may be transferred to one of the statements indicated, depending on the action specified by

49 the subroutine.

1 **12.4.5 Elemental Intrinsic Subroutine Reference.** A reference to an elemental intrinsic subrou-  
 2 tine is an elemental reference if all INTENT (OUT) and INTENT (INOUT) <sup>actual arguments</sup> ~~actual~~ arguments are  
 3 arrays that have the same shape and the remaining actual arguments are conformable with them.  
 4 The values of the elements of the INTENT (OUT) and INTENT (INOUT) <sup>dummy</sup> ~~arrays~~ are the same as  
 5 would be obtained if the subroutine were applied separately to corresponding <sup>elements</sup> ~~elements~~ of each  
 6 argument.

## 7 12.5 Procedure Definition.

8 **12.5.1 Intrinsic Procedure Definition.** Intrinsic procedures are defined as an inherent part of the  
 9 processor. A standard-conforming processor must include the intrinsic procedures described in  
 10 Section 13, but may include others. However, a standard-conforming program must not make use  
 11 of intrinsic procedures other than those described in Section 13.

12 **12.5.2 Procedures Defined by Subprograms.** When a procedure defined by a subprogram is  
 13 invoked, an instance (12.5.2.4) of the procedure is created and executed. Execution begins with the  
 14 first executable construct following the FUNCTION, SUBROUTINE, or ENTRY statement specify-  
 15 ing the name of the procedure invoked or with the END statement if there is no other executable  
 16 construct.

17 **12.5.2.1 Effects of INTENT Attribute on Subprograms.** The INTENT attribute of dummy data  
 18 objects limits the way in which they may be used in a subprogram. A dummy data object having  
 19 INTENT (IN) must not be defined or redefined by the subprogram. A dummy data object having  
 20 INTENT (OUT) is initially undefined in the subprogram. A dummy data object with INTENT  
 21 (INOUT) may be referenced or be defined. A dummy data object whose intent is not specified is  
 22 subject to the limitations of the data entity that is the associated actual argument. That is, a refer-  
 23 ence to the dummy data object may occur if the actual argument is defined and the dummy data  
 24 object may be defined if the actual argument is definable.

25 **12.5.2.2 Function Subprogram.** A function subprogram is a subprogram that has a FUNCTION  
 26 statement as its first statement.

27 R1215 *function-subprogram* is *function-stmt*  
 28 [ *specification-part* ]  
 29 [ *execution-part* ]  
 30 [ *internal-subprogram-part* ]  
 31 *end-function-stmt*

32 R1216 *function-stmt* is [ *prefix* ] FUNCTION *function-name* ■  
 33 ■ ( [ *dummy-arg-name-list* ] ) [ RESULT ( *result-name* ) ]

34 Constraint: <sup>If RESULT is specified</sup> The *function-name*, ~~if different from the result-name,~~ must not appear in any specifica-  
 35 tion statement in the scoping unit of the function subprogram.

36 R1217 *prefix* is *type-spec* [ RECURSIVE ]  
 37 or RECURSIVE [ *type-spec* ]

38 R1218 *end-function-stmt* is END [ FUNCTION [ *function-name* ] ]

39 Constraint: If RESULT is specified, *result-name* must not be the same as *function-name*.

40 Constraint: FUNCTION must be present on the *end-function-stmt* of an internal or module func-  
 41 tion.

42 Constraint: An internal function must not contain an ENTRY statement.

43 Constraint: An internal function must not contain an *internal-subprogram-part*.

44 Constraint: If a *function-name* is present on the *end-function-stmt*, it must be identical to the  
 45 *function-name* specified in the *function-stmt*.

## Procedures

- 1 The type and type parameters (if any) of the result of the function defined by a function subprogram may be specified by a type specification in the FUNCTION statement or by the name of the result variable appearing in a type statement in the declaration part of the function subprogram. It 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 a pointer, this must be specified by specifications of the name of the result variable within the function body. The specifications of the function result attributes, the specification of dummy argument attributes, and the information in the procedure heading collectively define the interface of the function (12.3).
- 10 The keyword **RECURSIVE** must be present if the function directly or indirectly invokes itself or a function defined by an **ENTRY** statement in the same subprogram. Similarly, **RECURSIVE** must be present if a function defined by an **ENTRY** statement in the subprogram directly or indirectly invokes itself, another function defined by an **ENTRY** statement in that subprogram, or the function defined by the **FUNCTION** statement.
- 15 The name of the function is *function-name*.
- 16 If **RESULT** is specified, the name of the result variable of the function is *result-name*, its characteristics (12.2.2) are those of the function result, and all occurrences of the function name in *execution-part* statements in the scoping unit are recursive function references. If **RESULT** is not specified, the result variable is *function-name* and all occurrences of the function name in *execution-part* statements in the scoping unit are references to the result variable. The value of the result variable at the completion of execution of the function is the value returned by the function. If the function result has been declared to be a pointer, the shape of the value returned by the function is determined by the shape of the result variable when the execution of the function is completed. If the result variable is not a pointer, its value must be defined by the function. If the function result has been declared a pointer, the function must either associate a target with the result variable pointer or cause the association status of this pointer to become defined as disassociated.
- 27 If both **RECURSIVE** and **RESULT** are specified, the interface of the function being defined is explicit within the function subprogram.

29 An example of a recursive function is:

```
30 RECURSIVE FUNCTION CUMM_SUM (ARRAY) RESULT (C_SUM)
31   REAL ARRAY (:), CUMM_SUM (SIZE (ARRAY))
32   ! The characteristics of C_SUM are those of CUMM_SUM.
33   INTENT (IN) ARRAY
34   INTEGER N
35   N = SIZE (ARRAY)
36   IF (N .LE. 1) THEN
37     C_SUM = ARRAY
38   ELSE
39     N = N / 2
40     C_SUM (:N) = CUMM_SUM (ARRAY (:N))
41     C_SUM (N+1:) = C_SUM (N) + CUMM_SUM (ARRAY (N+1:))
42   END IF
43 END FUNCTION CUMM_SUM
```

44 **12.5.2.3 Subroutine Subprogram.** A subroutine subprogram is a subprogram that has a **SUBROUTINE** statement as its first statement.

```
46 R1219 subroutine-subprogram is subroutine-stmt
47     [ specification-part ]
48     [ execution-part ]
49     [ internal-subprogram-part ]
50     end-subroutine-stmt
51 R1220 subroutine-stmt is [ RECURSIVE ] SUBROUTINE subroutine-name ■
52     ■ [ ( [ dummy-arg-list ] ) ]
```

- 1 R1221 *dummy-arg* is *dummy-arg-name*  
 2 or \*
- 3 R1222 *end-subroutine-stmt* is END [ SUBROUTINE [ *subroutine-name* ] ]
- 4 Constraint: SUBROUTINE must be present on the *end-subroutine-stmt* of an internal or module  
 5 subroutine.
- 6 Constraint: An internal subroutine must not contain an ENTRY statement.
- 7 Constraint: An internal subroutine must not contain an *internal-subprogram-part*.
- 8 Constraint: If a *subroutine-name* is present on the *end-subroutine-stmt*, it must be identical to the  
 9 *subroutine-name* specified in the *subroutine-stmt*.
- 10 The keyword RECURSIVE must be present if the subroutine directly or indirectly invokes itself or  
 11 a subroutine defined by an ENTRY statement in the same subprogram. Similarly, RECURSIVE  
 12 must be present if a subroutine defined by an ENTRY statement in the subprogram directly or  
 13 indirectly invokes itself, another subroutine defined by an ENTRY statement in that subprogram,  
 14 or the subroutine defined by the SUBROUTINE statement.
- 15 If RECURSIVE is specified, the interface of the subroutine being defined is explicit within the sub-  
 16 routine subprogram.
- 17 The name of the subroutine is *subroutine-name*.
- 18 **12.5.2.4 Instances of a Subprogram.** When a function or subroutine defined by a subprogram  
 19 is invoked, an instance of that subprogram is created.
- 20 Each instance has an independent sequence of execution and an independent set of dummy argu-  
 21 ments and local nonsaved data objects. If an internal procedure or statement function contained in  
 22 the subprogram is invoked directly from an instance of the subprogram or an internal procedure  
 23 or statement function that has access to the entities of that instance, the created instance of that  
 24 internal procedure or statement function also has access to the entities of that instance of the host  
 25 subprogram.
- 26 All other entities are shared by all instances of the subprogram. For example, the value of a saved  
 27 data object appearing in one instance may have been defined in a previous instance or by initiali-  
 28 zation in a DATA statement or type declaration. *statement*
- 29 **12.5.2.5 ENTRY Statement.** An ENTRY statement permits a procedure reference to begin with a  
 30 particular executable statement within the function or subroutine subprogram in which the  
 31 ENTRY statement appears.
- 32 R1223 *entry-stmt* is ENTRY *entry-name* [ ( [ *dummy-arg-list* ] ) ] ■  
 33 Constraint: If RESULT is specified, ■ [ RESULT ( *result-name* ) ]  
 the *entry-name* must not appear in any specification statement in the *subprogram* or the *subprogram*.
- 34 Constraint: An *entry-stmt* may appear only in an *external-subprogram* or *module-subprogram*. An  
 35 *entry-stmt* must not appear within an *executable-construct*.
- 36 Constraint: RESULT may be present only if the *entry-stmt* is contained in a function subprogram.
- 37 Constraint: Within the subprogram containing the *entry-stmt*, the *entry-name* must not appear as  
 38 a dummy argument in the FUNCTION or SUBROUTINE statement or in another  
 39 ENTRY statement and it must not appear in an EXTERNAL statement.
- 40 Constraint: A *dummy-arg* may be an alternate return indicator only if the ENTRY statement is contained in a subrou-  
 41 tine subprogram.
- 42 Constraint: If RESULT is specified, *result-name* must not be the same as *entry-name*.
- 43 Optionally, a subprogram may have one or more ENTRY statements.
- 44 If the ENTRY statement is contained in a function subprogram, an additional function is defined  
 45 by that subprogram. The name of the function is *entry-name* and its result variable is *result-name* or  
 46 is *entry-name* if no *result-name* is provided. The characteristics of the function result are specified

## Procedures

- 1 by specifications of *entry-name*. The dummy arguments of the function are those specified on the  
2 ENTRY statement. If the characteristics of the result of the function named on the ENTRY state-  
3 ment are the same as the characteristics of the function named on the FUNCTION statement, their  
4 result variables identify the same variable. Otherwise, they are storage associated and must all be  
5 ~~scalar~~ and all be of type default character with identical length or all be ~~scalar~~ and one of the types  
6 ~~default integer, default real, double precision real, default complex, or default logical.~~  
*scalars without the PSEUDO attribute.*
- 7 If RESULT is specified on the ENTRY statement and RECURSIVE is specified on the FUNCTION  
8 statement, the interface of the function defined by the ENTRY statement is explicit within the func-  
9 tion subprogram.
- 10 If the ENTRY statement is contained in a subroutine subprogram, an additional subroutine is  
11 defined by that subprogram. The name of the subroutine is *entry-name*. The dummy arguments of  
12 the subroutine are those specified on the ENTRY statement.
- 13 If RECURSIVE is specified on the SUBROUTINE statement, the interface of the subroutine defined  
14 by the ENTRY statement is explicit within the subroutine subprogram.
- 15 The order, number, types, kind type parameters, and names of the dummy arguments in an  
16 ENTRY statement may differ from the order, number, types, kind type parameters, and names of  
17 the dummy arguments in the FUNCTION or SUBROUTINE statement in the containing program.
- 18 Because an ENTRY statement defines an additional function or an additional subroutine, it is refer-  
19 enced in the same manner as any other function or subroutine (12.4).
- 20 In a subprogram, a name that appears as a dummy argument in an ENTRY statement must not  
21 appear in an executable statement preceding that ENTRY statement, unless it also appears in a  
22 FUNCTION, SUBROUTINE, or ENTRY statement that precedes the executable statement.
- 23 In a subprogram, a name that appears as a dummy argument in an ENTRY statement must not  
24 appear in the expression of a statement function unless the name is also a dummy argument of the  
25 statement function, appears in a FUNCTION or SUBROUTINE statement, or appears in an ENTRY  
26 statement that precedes the statement function statement.
- 27 If a dummy argument appears in an executable statement, the execution of the executable state-  
28 ment is permitted during the execution of a reference to the function or subroutine only if the  
29 dummy argument appears in the dummy argument list of the procedure name referenced and it is  
30 present if it is an OPTIONAL argument (12.5.2.8).
- 31 A scoping unit containing a reference to a procedure defined by an ENTRY statement may have  
32 access to an interface body for the procedure. The procedure header for the interface body must be  
33 a FUNCTION statement for an entry in a function subprogram and must be a SUBROUTINE state-  
34 ment for an entry in a subroutine subprogram.
- 35 The keyword RECURSIVE is not used in an ENTRY statement. Instead, the presence or absence of  
36 RECURSIVE on the initial SUBROUTINE or FUNCTION statement controls whether the proce-  
37 dure defined by an ENTRY statement is permitted to reference itself.

### 38 12.5.2.6 RETURN Statement.

39 R1224 *return-stmt* is RETURN [ *scalar-int-expr* ]

40 Constraint: The *return-stmt* must be contained in the scoping unit of a function or subroutine sub-  
41 program.

42 Constraint: The *scalar-int-expr* is allowed only in the scoping unit of a subroutine subprogram.

43 Execution of the RETURN statement completes execution of the instance of the subprogram in  
44 which it appears. If the expression is present and has a value *n* between 1 and the number of asterisks in the dummy  
45 argument list, the CALL statement that invoked the subroutine transfers control to the statement identified by the *n*th alter-  
46 nate return specifier in the actual argument list. If the expression is omitted or has a value outside the required range, there  
47 is no transfer of control to an alternate return.

48 Execution of an *end-function-stmt* or *end-subroutine-stmt* is equivalent to executing a RETURN state-  
49 ment with no expression.



1 **12.5.2.7 CONTAINS Statement.**2 R1225 *contains-stmt* is CONTAINS3 The CONTAINS statement separates the body of a main program, module, or subprogram from  
4 any internal or module subprograms it may contain. The CONTAINS statement is not executable.5 **12.5.2.8 Restrictions on Dummy Arguments Not Present.** A dummy argument is present in an  
6 instance of a subprogram if it is associated with an actual argument and the actual argument either  
7 is a dummy argument that is present in the invoking procedure or is not a dummy argument of  
8 the invoking procedure. A dummy argument that is not optional must be present. An optional  
9 dummy argument that is not present is subject to the following restrictions:

10 (1) If it is a dummy data object, it must not be referenced or be defined.

11 (2) If it is a dummy procedure, it must not be invoked.

12 (3) It must not be supplied as an actual argument corresponding to a nonoptional dummy  
13 argument other than the argument of the PRESENT intrinsic function.14 (4) It may be supplied as an actual argument corresponding to an optional dummy argu-  
15 ment. The optional dummy argument is then also considered not to be associated with  
16 an actual argument.17 **12.5.2.9 Restrictions on Entities Associated with Dummy Arguments.** While an entity is asso-  
18 ciated with a dummy argument, the following restrictions hold:19 (1) No action may be taken that affects the value or availability of the entity or any part of  
20 it, except through the dummy argument. For example, in

```

21 SUBROUTINE OUTER
22     REAL, POINTER :: A (:)
23     ...
24     ALLOCATE (A (1:N))
25     ...
26     CALL INNER (A)
27     ...
28 CONTAINS
29     SUBROUTINE INNER (B)
30     REAL :: B (:)
31     ...
32 END SUBROUTINE INNER

33 SUBROUTINE SET (C, D)
34     REAL, INTENT (OUT) :: C
35     REAL, INTENT (IN) :: D
36     C = D
37 END SUBROUTINE SET
38 END SUBROUTINE OUTER

```

39 an assignment statement such as

40 A (1) = 1.0

41 would not be permitted during the execution of INNER because this would be changing  
42 A without using B, but statements such as

43 B (1) = 1.0

44 or

45 CALL SET (B (1), 1.0)

46 would be allowed. Similarly,

## Procedures

1 DEALLOCATE (A)  
2 would not be allowed because this affects the availability of A without using B. In this  
3 case,  
4 DEALLOCATE (B)  
5 also would not be permitted, but would be permitted if B were declared with the  
6 POINTER attribute.

7 Note that if there is a partial or complete overlap between the actual arguments associ-  
8 ated with two different dummy arguments of the same procedure, the overlapped por-  
9 tions must not be defined, redefined, or become undefined during the execution of the  
10 procedure. For example, in

```
11 CALL SUB (A (1:5), A (3:9))
```

12 A (3:5) must not be defined, redefined, or become undefined through the first dummy  
13 argument because it is part of the argument associated with the second dummy argu-  
14 ment and must not be defined, redefined, or become undefined through the second  
15 dummy argument because it is part of the argument associated with the first dummy  
16 argument. A (1:2) remains definable through the first dummy argument and A (6:9)  
17 remains definable through the second dummy argument.

18 This restriction applies equally to pointer targets. For example, in

```
19 REAL, DIMENSION (10), TARGET :: A  
20 REAL, DIMENSION (:), POINTER :: B, C  
21 B => A (1:5)  
22 C => A (3:9)  
23 CALL SUB (B, C)
```

24 B (3:5) cannot be defined because it is part of the argument associated with the second  
25 dummy argument. C (1:3) cannot be defined because it is part of the argument associ-  
26 ated with the first dummy argument. A (1:2) [which is B (1:2)] remains definable  
27 through the first dummy argument and A (6:9) [which is C (4:7)] remains definable  
28 through the second dummy argument.

29 Note that since a dummy argument declared with an intent of IN cannot be used to  
30 change the associated actual argument, the associated actual argument remains con-  
31 stant throughout the execution of the procedure.

- 32 (2) If any part of the entity is defined through the dummy argument, then at any time dur-  
33 ing the execution of the procedure, either before or after the definition, it may be refer-  
34 enced only through that dummy argument. For example, in

```
35 MODULE DATA  
36     REAL :: W, X, Y, Z  
37 END MODULE DATA
```

```
38 PROGRAM MAIN  
39     USE DATA  
40     ...  
41     CALL INIT (X)  
42     ...  
43 END PROGRAM MAIN
```

```
44 SUBROUTINE INIT (V)  
45     USE DATA  
46     ...  
47     READ (*, *) V  
48     ...  
49 END SUBROUTINE INIT
```

1 variable X must not be directly referenced at any time during the execution of INIT  
 2 because it is being defined through the dummy argument V. X may be (indirectly) ref-  
 3 erenced through V. W, Y, and Z may be directly referenced. X may, of course, be  
 4 directly referenced once execution of INIT is complete.

5 **12.5.3 Definition of Procedures by Means Other Than Fortran.** The means other than Fortran  
 6 by which a procedure may be defined are processor dependent. A reference to such a procedure is  
 7 made as though it were defined by an external subprogram. The definition of a non-Fortran proce-  
 8 dure must not be contained in a Fortran program unit and a Fortran program unit must not be  
 9 contained in the definition of a non-Fortran procedure. The interface to a non-Fortran procedure  
 10 may be specified in an interface block.

11 **12.5.4 Statement Function.** A statement function is a function defined by a single statement.

12 R1226 *stmt-function-stmt* is *function-name* ([ *dummy-arg-name-list* ]) = *scalar-expr*

13 Constraint: The *scalar-expr* may be composed only of constants (literal and named), references to  
 14 scalar variables and array elements, references to functions and function dummy  
 15 procedures, and intrinsic operators. If a reference to ~~another~~ statement function  
 16 appears in *scalar-expr*, its definition must have been provided earlier in the scoping  
 17 unit, and must not be the name of the statement function being defined.

18 Constraint: Named constants in *scalar-expr* must have been declared earlier in the scoping unit. If  
 19 array elements appear in *scalar-expr*, the parent array must have been declared as an  
 20 array earlier in the scoping unit. If a scalar variable, array element, function refer-  
 21 ence, or dummy function reference is typed by the implicit typing rules, its appear-  
 22 ance in any subsequent type declaration statement must confirm this implied type  
 23 and the values of any implied type parameters.

24 Constraint: The *function-name* and each *dummy-arg-name* must be specified, explicitly or implic-  
 25 itly, to be scalar data objects.

26 Constraint: A given *dummy-arg-name* may appear only once in any *dummy-arg-name-list*.

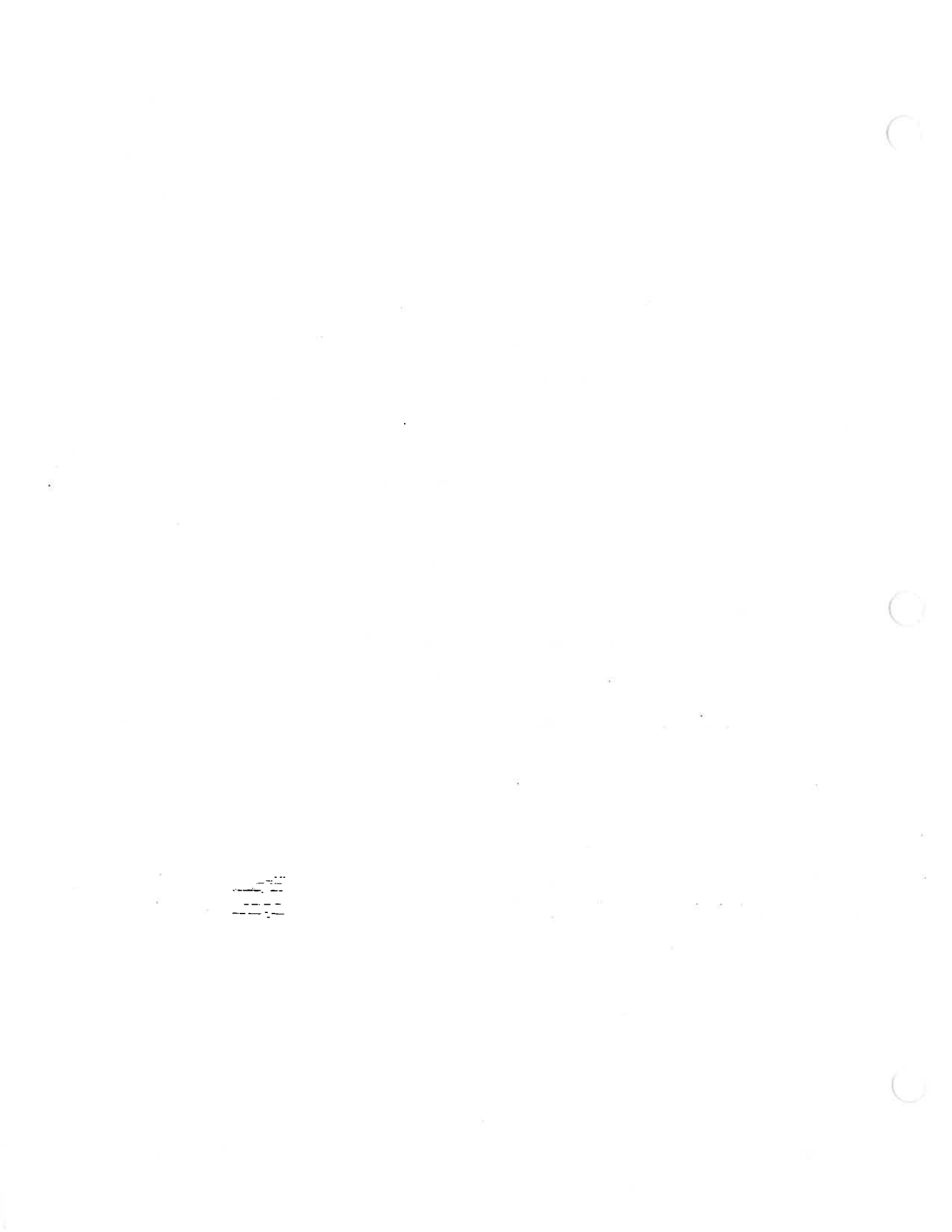
27 Constraint: Each scalar variable reference in *scalar-expr* may be either a reference to a dummy  
 28 argument of the statement function or a reference to a variable ~~within~~ the same scop-  
 29 ing unit as the statement function statement. *local to*

30 The dummy arguments have a scope of the statement function statement. Each dummy argument  
 31 has the same type and type parameters as the entity of the same name in the scoping unit contain-  
 32 ing the statement function.

33 A statement function must not be supplied as a procedure argument.

34 The value of a statement function reference is obtained by evaluating the expression using the val-  
 35 ues of the actual arguments for the values of the corresponding dummy arguments and, if neces-  
 36 sary, converting the result to the declared type and type attributes of the function.

37 A function reference in the scalar expression must not cause a dummy argument of the statement  
 38 function to become redefined or undefined.



## 13. INTRINSIC PROCEDURES

1 There are four classes of intrinsic procedures: inquiry functions, elemental functions, transforma-  
2 tional functions, and subroutines. One intrinsic subroutine is elemental.

3 **13.1 Intrinsic Functions.** An intrinsic function is an inquiry function, an elemental function,  
4 or a transformational function. An inquiry function is one whose result depends on the proper-  
5 ties of its principal argument other than the value of this argument; in fact, the argument value  
6 may be undefined. An elemental function is one that is specified for scalar arguments, but may be  
7 applied to array arguments as described in 13.2. All other intrinsic functions are **transformational**  
8 **functions**; they almost all have one or more array-valued arguments or an array-valued result.

9 Generic names of intrinsic functions are listed in 13.10. In most cases, generic functions accept  
10 arguments of more than one type and the type of the result is the same as the type of the argu-  
11 ments. Specific names of intrinsic functions with corresponding generic names are listed in 13.12.

12 If an intrinsic function is used as an actual argument to a procedure, its specific name must be used  
13 and it may be referenced in the called procedure only with scalar arguments. If an intrinsic func-  
14 tion does not have a specific name, it must not be used as an actual argument (12.4.1.2).

### 15 13.2 Elemental Intrinsic Procedures.

16 **13.2.1 Elemental Intrinsic Function Arguments and Results.** If a generic name or a specific  
17 name is used to reference an elemental intrinsic function, the shape of the result is the same as the  
18 shape of the argument with the greatest rank. If the arguments are all scalar, the result is scalar.  
19 For those elemental intrinsic functions that have more than one argument, all arguments must be  
20 conformable. In the array-valued case, the values of the elements, if any, of the result are the same  
21 as would have been obtained if the scalar-valued function had been applied separately, in any  
22 order, to corresponding elements of each argument. Arguments called KIND must always be  
23 specified as scalar integer initialization expressions.

24 **13.2.2 Elemental Intrinsic Subroutine Arguments.** An elemental subroutine is one that is speci-  
25 fied for scalar arguments, but may be applied to array arguments. In a reference to an elemental  
26 intrinsic subroutine, either all actual arguments must be scalar, or all INTENT (OUT) and INTENT  
27 (INOUT) arguments must be arrays of the same shape and the remaining arguments must be con-  
28 formable with them. In the case that the INTENT (OUT) arguments are arrays, the values of the  
29 elements, if any, of the results are the same as would be obtained if the subroutine with scalar  
30 arguments were applied separately, in any order, to corresponding elements of each argument.

31 **13.3 Positional Arguments or Argument Keywords.** All intrinsic procedures may be  
32 invoked with either positional arguments or argument keywords. The descriptions in 13.13 give  
33 the keyword names and positional sequence. A keyword is required for an argument only if a pre-  
34 ceding optional argument is omitted.

35 **13.4 Argument Presence Inquiry Function.** The inquiry function PRESENT permits an  
36 inquiry to be made about the presence of an actual argument associated with a dummy argument  
37 that has the OPTIONAL attribute.

### 38 13.5 Numeric, Mathematical, Character, Kind, Logical, and Bit Procedures.

39 **13.5.1 Numeric Functions.** The elemental functions INT, REAL, DBLE, and CMPLX perform  
40 type conversions. The elemental functions AIMAG, CONJG, AINT, ANINT, NINT, ABS, MOD,  
41 SIGN, DIM, DPROD, MODULO, FLOOR, CEILING, MAX, and MIN perform simple numeric  
42 operations.

## Intrinsic Procedures

1 **13.5.2 Mathematical Functions.** The elemental functions SQRT, EXP, LOG, LOG10, SIN, COS,  
2 TAN, ASIN, ACOS, ATAN, ATAN2, SINH, COSH, and TANH evaluate elementary mathematical  
3 functions.

4 **13.5.3 Character Functions.** The elemental functions ICHAR, CHAR, LGE/LGT, LLE, LLT,  
5 IACHAR, ACHAR, INDEX, VERIFY, ADJUSTL, ADJUSTR, SCAN, and LEN/TRIM perform char-  
6 acter operations. The transformational function REPEAT returns repeated concatenations of a  
7 character string argument. The transformational function TRIM returns the argument with trailing  
8 blanks removed.

9 **13.5.4 Character Inquiry Function.** The inquiry function LEN returns the length of a character  
10 entity. The value of the argument to this function need not be defined. It is not necessary for a  
11 processor to evaluate the argument of this function if the value of the function can be determined  
12 otherwise.

13 **13.5.5 Kind Functions.** The inquiry function KIND returns the kind type parameter value of an  
14 integer, real, complex, logical, or character entity. The value of the argument to this function need  
15 not be defined. The transformational function SELECTED\_REAL\_KIND returns the real kind type  
16 parameter value that has at least the decimal precision and exponent range specified by its argu-  
17 ments. The transformational function SELECTED\_INT\_KIND returns the integer kind type  
18 parameter value that has at least the decimal exponent range specified by its argument.

19 **13.5.6 Logical Function.** The elemental function LOGICAL converts between objects of type log-  
20 ical with different kind type parameter values.

21 **13.5.7 Bit Manipulation and Inquiry Procedures.** The bit manipulation procedures consist of a  
22 set of ten functions and one subroutine. Logical operations on bits are provided by the functions  
23 IOR, IAND, NOT, and IEXOR; shift operations are provided by the functions ISHFT and ISHFTC;  
24 bit subfields may be referenced by the function IBITS and by the subroutine MVBITS; single-bit  
25 processing is provided by the functions BTEST, IBSET, and IBCLR.

26 For the purposes of these procedures, a bit is defined to be a binary digit  $w$  located at position  $k$  of  
27 a nonnegative integer scalar object based on a model nonnegative integer defined by

$$j = \sum_{k=0}^{s-1} w_k \times 2^k$$

30 and for which  $w_k$  may have the value 0 or 1. An example of a model number compatible with the  
31 examples used in 13.7.1 would have  $s = 32$ , thereby defining a 32-bit integer.

32 An inquiry function BIT\_SIZE is available to determine the parameter  $s$  of the model. The value of  
33 the argument of this function need not be defined. It is not necessary for a processor to evaluate  
34 the argument of this function if the value of the function can be determined otherwise.

35 Effectively, this model defines an integer object to consist of  $s$  bits in sequence numbered from  
36 right to left from 0 to  $s - 1$ . This model is valid only in the context of the use of such an object as  
37 the argument or result of one of the bit manipulation procedures. In all other contexts, the model  
38 defined for an integer in 13.7.1 applies. In particular, whereas the models are identical for  $w_{s-1} = 0$ ,  
39 they do not correspond for  $w_{s-1} = 1$  and the interpretation of bits in such objects is processor  
40 dependent.

41 **13.6 Transfer Function.** The function TRANSFER specifies that the physical representation of  
42 the first argument is to be treated as if it were one of the type and type parameters of the second  
43 argument with no conversion.

1 **13.7 Numeric Manipulation and Inquiry Functions.** The numeric manipulation and  
 2 inquiry functions are described in terms of a model for the representation and behavior of num-  
 3 bers on a processor. The model has parameters which are determined so as to make the model  
 4 best fit the machine on which the executable program is executed.

5 **13.7.1 Models for Integer and Real Data.** The model set for integer  $i$  is defined by:

$$i = s \times \sum_{k=1}^q w_k \times r^{k-1}$$

8 where  $r$  is an integer exceeding one,  $q$  is a positive integer, each  $w_k$  is a nonnegative integer less  
 9 than  $r$ , and  $s$  is +1 or -1. The model set for real  $x$  is defined by:

$$x = \begin{cases} 0 \text{ or} \\ s \times b^e \times \sum_{k=1}^p f_k \times b^{-k}, \end{cases}$$

12 where  $b$  and  $p$  are integers exceeding one; each  $f_k$  is a nonnegative integer less than  $b$ , except  $f_1$   
 13 which is also nonzero;  $s$  is +1 or -1; and  $e$  is an integer that lies between some integer maximum  
 14  $e_{\max}$  and some integer minimum  $e_{\min}$  inclusively. For  $x = 0$ , its exponent  $e$  and digits  $f_k$  are defined  
 15 to be zero. The integer parameters  $r$  and  $q$  determine the set of model integers and the integer  
 16 parameters  $b$ ,  $p$ ,  $e_{\min}$ , and  $e_{\max}$  determine the set of model floating point numbers. The parameters  
 17 of the integer and real models are available for each integer and real data type implemented by the  
 18 processor. The parameters characterize the set of available numbers in the definition of the model.  
 19 The numeric manipulation and inquiry functions provide values related to the parameters and  
 20 other constants related to them. Examples of these functions in this section use the models:

$$i = s \times \sum_{k=1}^{31} w_k \times 2^{k-1}$$

23 and

$$x = 0 \text{ or } s \times 2^e \times \left[ 1/2 + \sum_{k=2}^{24} f_k \times 2^{-k} \right], \quad -126 \leq e \leq 127$$

26 **13.7.2 Numeric Inquiry Functions.** The inquiry functions RADIX, DIGITS, MINEXPONENT,  
 27 MAXEXPONENT, PRECISION, RANGE, HUGE, TINY, and EPSILON return scalar values related  
 28 to the parameters of the model associated with the type and type parameters of the arguments.  
 29 The value of the arguments to these functions need not be defined, pointer arguments may be dis-  
 30 associated, and array arguments need not be allocated.

31 **13.7.3 Floating Point Manipulation Functions.** The elemental functions EXPONENT, SCALE,  
 32 NEAREST\_FRACTION, SET\_EXPONENT, SPACING, and RRSPACING return values related to  
 33 the components of the model values (13.7.1) associated with the actual values of the arguments.

34 **13.8 Array Intrinsic Functions.** The array intrinsic functions perform the following operations  
 35 on arrays: vector and matrix multiplication, numeric or logical computation that reduces the rank,  
 36 array structure inquiry, array construction, array manipulation, and geometric location.

37 **13.8.1 The Shape of Array Arguments.** The transformational array intrinsic functions operate  
 38 on each array argument as a whole. The shape of the corresponding actual argument must there-  
 39 fore be defined; that is, the actual argument must be an array section, an assumed-shape array, an  
 40 explicit-shape array, a pointer that is associated with a target, an allocatable array that has been  
 41 allocated, or an array-valued expression. It must not be an assumed-size array.

## Intrinsic Procedures

- 1 Some of the inquiry intrinsic functions accept array arguments for which the shape need not be
- 2 defined. Assumed-size arrays may be used as arguments to these functions; they include the func-
- 3 tion LBOUND and certain references to SIZE and UBOUND.
  
- 4 **13.8.2 Mask Arguments.** Some array intrinsic functions have an optional MASK argument that
- 5 is used by the function to select the elements of one or more arguments to be operated on by the
- 6 function. Any element not selected by the mask need not be defined at the time the function is
- 7 invoked.
  
- 8 The MASK affects only the value of the function, and does not affect the evaluation, prior to invok-
- 9 ing the function, of arguments that are array expressions.
  
- 10 A MASK argument must be of type logical.
  
- 11 **13.8.3 Vector and Matrix Multiplication Functions.** The matrix multiplication function
- 12 MATMUL operates on two matrices, or on one matrix and one vector, and returns the corre-
- 13 sponding matrix-matrix, matrix-vector, or vector-matrix product. The arguments to MATMUL
- 14 may be numeric (integer, real, or complex) or logical arrays. On logical matrices and vectors,
- 15 MATMUL performs Boolean matrix multiplication.
  
- 16 The dot product function DOT\_PRODUCT operates on two vectors and returns their scalar prod-
- 17 uct. The vectors are of the same type (numeric or logical) as for MATMUL. For logical vectors,
- 18 DOT\_PRODUCT returns the Boolean scalar product.
  
- 19 **13.8.4 Array Reduction Functions.** The array reduction functions SUM, PRODUCT, MAXVAL,
- 20 MINVAL, COUNT, ANY, and ALL perform numerical, logical, and counting operations on arrays.
- 21 They may be applied to the whole array to give a scalar result or they may be applied over a given
- 22 dimension to yield a result of rank reduced by one. By use of a logical mask that is conformable
- 23 with the given array, the computation may be confined to any subset of the array (for example, the
- 24 positive elements).
  
- 25 **13.8.5 Array Inquiry Functions.** The function ALLOCATED returns a value true if the array
- 26 argument is currently allocated, and returns false otherwise. The functions SIZE, SHAPE,
- 27 LBOUND, and UBOUND return, respectively, the size of the array, the shape, and the lower and
- 28 upper bounds of the subscripts along each dimension. The size, shape, or bounds must be defined.
  
- 29 The values of the array arguments to these functions need not be defined.
  
- 30 **13.8.6 Array Construction Functions.** The functions MERGE, SPREAD, PACK, and UNPACK
- 31 construct new arrays from the elements of existing arrays. MERGE combines two conformable
- 32 arrays into one array by an element-wise choice based on a logical mask. SPREAD constructs an
- 33 array from several copies of an actual argument (SPREAD does this by adding an extra dimension,
- 34 as in forming a book from copies of one page). PACK and UNPACK respectively gather and scat-
- 35 ter the elements of a one-dimensional array from and to positions in another array where the posi-
- 36 tions are specified by a logical mask.
  
- 37 **13.8.7 Array Reshape Function.** RESHAPE produces an array with the same elements and a dif-
- 38 ferent shape.
  
- 39 **13.8.8 Array Manipulation Functions.** The functions TRANSPOSE, EOSHIFT, and CSHIFT
- 40 manipulate arrays. TRANSPOSE performs the matrix transpose operation on a two-dimensional
- 41 array. The shift functions leave the shape of an array unaltered but shift the positions of the ele-
- 42 ments parallel to a specified dimension of the array. These shifts are either circular (CSHIFT), in
- 43 which case elements shifted off one end reappear at the other end, or end-off (EOSHIFT), in which
- 44 case specified boundary elements are shifted into the vacated positions.



1 **13.8.9 Array Location Functions.** The functions MAXLOC and MINLOC return the location  
 2 (subscripts) of an element of an array that has maximum and minimum values, respectively. By  
 3 use of an optional logical mask that is conformable with the given array, the reduction may be con-  
 4 fined to any subset of the array.

5 **13.8.10 Pointer Association Status Inquiry Functions.** The function ASSOCIATED tests  
 6 whether a pointer is currently associated with any target, with a particular target, or with the same  
 7 target as another pointer.

8 **13.9 Intrinsic Subroutines.** Intrinsic subroutines are supplied by the processor and have the  
 9 special definitions given in 13.11 and 13.13. An intrinsic subroutine is referenced by a CALL state-  
 10 ment that uses its name explicitly. The name of an intrinsic subroutine must not be used as an  
 11 actual argument. The effect of a subroutine reference is as specified in 13.13.

12 **13.9.1 Date and Time Subroutines.** The subroutines DATE\_AND\_TIME and SYSTEM\_CLOCK  
 13 return data from the date and real-time clock. The time returned is local, but there are facilities for  
 14 finding out the difference between local time and Coordinated Universal Time.

15 **13.9.2 Pseudorandom Numbers.** The subroutine RANDOM\_NUMBER returns a pseudoran-  
 16 dom number or an array of pseudorandom numbers. The subroutine RANDOM\_SEED initializes  
 17 or restarts the pseudorandom number sequence.

18 **13.9.3 Bit Copy Subroutine.** The elemental subroutine MVBITS copies a bit field from a specified  
 19 position in one integer object to a specified position in another.

20 **13.10 Generic Intrinsic Functions.** For all of the intrinsic procedures, the arguments shown  
 21 are the names that must be used for keywords when using the keyword form for actual argu-  
 22 ments. For example, a reference to CMPLX may be written in the form CMPLX (A, B, M) or in the  
 23 form CMPLX (Y = B, KIND = M, X = A).

24 Many of the argument keywords have names that are indicative of their usage. For example:

|    |                  |                                             |
|----|------------------|---------------------------------------------|
| 25 | KIND             | Describes the KIND of the result            |
| 26 | STRING, STRING_A | An arbitrary character string               |
| 27 | BACK             | Indicates a string scan is                  |
| 28 |                  | to be from right to left (backward)         |
| 29 | MASK             | A mask that may be applied to the arguments |
| 30 | DIM              | A selected dimension of an array argument   |

31 **13.10.1 Argument Presence Inquiry Function.**

|    |             |                   |
|----|-------------|-------------------|
| 32 | PRESENT (A) | Argument presence |
|----|-------------|-------------------|

33 **13.10.2 Numeric Functions.**

|    |                    |                                               |
|----|--------------------|-----------------------------------------------|
| 34 | ABS (A)            | Absolute value                                |
| 35 | AIMAG (Z)          | Imaginary part of a complex number            |
| 36 | AINT (A, KIND)     | Truncation to whole number                    |
| 37 | Optional KIND      |                                               |
| 38 | ANINT (A, KIND)    | Nearest whole number                          |
| 39 | Optional KIND      |                                               |
| 40 | CEILING (A)        | Least integer greater than or equal to number |
| 41 | CMPLX (X, Y, KIND) | Conversion to complex type                    |
| 42 | Optional Y, KIND   |                                               |
| 43 | CONJG (Z)          | Conjugate of a complex number                 |
| 44 | DBLE (A)           | Conversion to double precision real type      |
| 45 | DIM (X, Y)         | Positive difference                           |

## Intrinsic Procedures

|    |                      |                                               |
|----|----------------------|-----------------------------------------------|
| 1  | DPROD (X, Y)         | Double precision real product                 |
| 2  | INT (A, KIND)        | Conversion to integer type                    |
| 3  | Optional KIND        |                                               |
| 4  | FLOOR (A)            | Greatest integer less than or equal to number |
| 5  | MAX (A1, A2, A3,...) | Maximum value                                 |
| 6  | Optional A3,...      |                                               |
| 7  | MIN (A1, A2, A3,...) | Minimum value                                 |
| 8  | Optional A3,...      |                                               |
| 9  | MOD (A, P)           | Remainder function                            |
| 10 | MODULO (A, P)        | Modulo function                               |
| 11 | NINT (A, KIND)       | Nearest integer                               |
| 12 | Optional KIND        |                                               |
| 13 | REAL (A, KIND)       | Conversion to real type                       |
| 14 | Optional KIND        |                                               |
| 15 | SIGN (A, B)          | Transfer of sign                              |

### 16 13.10.3 Mathematical Functions.

|    |              |                            |
|----|--------------|----------------------------|
| 17 | ACOS (X)     | Arccosine                  |
| 18 | ASIN (X)     | Arcsine                    |
| 19 | ATAN (X)     | Arctangent                 |
| 20 | ATAN2 (Y, X) | Arctangent                 |
| 21 | COS (X)      | Cosine                     |
| 22 | COSH (X)     | Hyperbolic cosine          |
| 23 | EXP (X)      | Exponential                |
| 24 | LOG (X)      | Natural logarithm          |
| 25 | LOG10 (X)    | Common logarithm (base 10) |
| 26 | SIN (X)      | Sine                       |
| 27 | SINH (X)     | Hyperbolic sine            |
| 28 | SQRT (X)     | Square root                |
| 29 | TAN (X)      | Tangent                    |
| 30 | TANH (X)     | Hyperbolic tangent         |

### 31 13.10.4 Character Functions.

|    |                                 |                                                                |
|----|---------------------------------|----------------------------------------------------------------|
| 32 | ACHAR (I)                       | Character in given position<br>in ASCII collating sequence     |
| 33 |                                 |                                                                |
| 34 | ADJUSTL (STRING)                | Adjust left                                                    |
| 35 | ADJUSTR (STRING)                | Adjust right                                                   |
| 36 | CHAR (I, KIND)                  | Character in given position<br>in processor collating sequence |
| 37 | Optional KIND                   |                                                                |
| 38 | IACHAR (C)                      | Position of a character<br>in ASCII collating sequence         |
| 39 |                                 |                                                                |
| 40 | ICHAR (C)                       | Position of a character<br>in processor collating sequence     |
| 41 |                                 |                                                                |
| 42 | INDEX (STRING, SUBSTRING, BACK) | Starting position of a substring                               |
| 43 | Optional BACK                   |                                                                |
| 44 | LEN_TRIM (STRING)               | Length without trailing blank characters                       |
| 45 | LGE (STRING_A, STRING_B)        | Lexically greater than or equal                                |
| 46 | LGT (STRING_A, STRING_B)        | Lexically greater than                                         |
| 47 | LLE (STRING_A, STRING_B)        | Lexically less than or equal                                   |
| 48 | LLT (STRING_A, STRING_B)        | Lexically less than                                            |
| 49 | REPEAT (STRING, NCOPIES)        | Repeated concatenation                                         |
| 50 | SCAN (STRING, SET, BACK)        | Scan a string for a character in a set                         |
| 51 | Optional BACK                   |                                                                |
| 52 | TRIM (STRING)                   | Remove trailing blank characters                               |
| 53 | VERIFY (STRING, SET, BACK)      | Verify the set of characters in a string                       |
| 54 | Optional BACK                   |                                                                |

|    |                                                        |                                                              |
|----|--------------------------------------------------------|--------------------------------------------------------------|
| 1  | <b>13.10.5 Character Inquiry Function.</b>             |                                                              |
| 2  | LEN (STRING)                                           | Length of a character entity                                 |
| 3  | <b>13.10.6 Kind Functions.</b>                         |                                                              |
| 4  | KIND (X)                                               | Kind type parameter value                                    |
| 5  | SELECTED_INT_KIND (R)                                  | Integer kind type parameter value,<br>given range            |
| 6  |                                                        |                                                              |
| 7  | SELECTED_REAL_KIND (P, R)                              | Real kind type parameter value,<br>given precision and range |
| 8  | <i>Optional P, R</i>                                   |                                                              |
| 9  | <b>13.10.7 Logical Function.</b>                       |                                                              |
| 10 | LOGICAL (L, KIND)                                      | Convert between objects                                      |
| 11 | Optional KIND                                          | of type logical with                                         |
| 12 |                                                        | different kind type parameters                               |
| 13 | <b>13.10.8 Numeric Inquiry Functions.</b>              |                                                              |
| 14 | DIGITS (X)                                             | Number of significant digits in the model                    |
| 15 | EPSILON (X)                                            | Number that is almost negligible compared to one             |
| 16 | HUGE (X)                                               | Largest number in the model                                  |
| 17 | MAXEXPONENT (X)                                        | Maximum exponent in the model                                |
| 18 | MINEXPONENT (X)                                        | Minimum exponent in the model                                |
| 19 | PRECISION (X)                                          | Decimal precision                                            |
| 20 | RADIX (X)                                              | Base of the model                                            |
| 21 | RANGE (X)                                              | Decimal exponent range                                       |
| 22 | TINY (X)                                               | Smallest positive number in the model                        |
| 23 | <b>13.10.9 Bit Inquiry Function.</b>                   |                                                              |
| 24 | BIT_SIZE (I)                                           | Number of bits in the model                                  |
| 25 | <b>13.10.10 Bit Manipulation Functions.</b>            |                                                              |
| 26 | BTEST (I, POS)                                         | Bit testing                                                  |
| 27 | IAND (I, J)                                            | Logical AND                                                  |
| 28 | IBCLR (I, POS)                                         | Clear bit                                                    |
| 29 | IBITS (I, POS, LEN)                                    | Bit extraction                                               |
| 30 | IBSET (I, POS)                                         | Set bit                                                      |
| 31 | IEOR (I, J)                                            | Exclusive OR                                                 |
| 32 | IOR (I, J)                                             | Inclusive OR                                                 |
| 33 | ISHFT (I, SHIFT)                                       | Logical shift                                                |
| 34 | ISHFTC (I, SHIFT, SIZE)                                | Circular shift                                               |
| 35 | Optional SIZE                                          |                                                              |
| 36 | NOT (I)                                                | Logical complement                                           |
| 37 | <b>13.10.11 Transfer Function.</b>                     |                                                              |
| 38 | TRANSFER (SOURCE, MOLD, SIZE)                          | Treat first argument as if                                   |
| 39 | Optional SIZE                                          | of type of second argument                                   |
| 40 | <b>13.10.12 Floating-point Manipulation Functions.</b> |                                                              |
| 41 | EXPONENT (X)                                           | Exponent part of a model number                              |
| 42 | FRACTION (X)                                           | Fractional part of a number                                  |
| 43 | NEAREST (X, S)                                         | Nearest different processor number in<br>given direction     |
| 44 |                                                        |                                                              |
| 45 | RRSPACING (X)                                          | Reciprocal of the relative spacing                           |

## Intrinsic Procedures

|    |                                                       |                                                        |
|----|-------------------------------------------------------|--------------------------------------------------------|
| 1  |                                                       | of model numbers near given number                     |
| 2  | SCALE (X, I)                                          | Multiply a real by its base to an integer power        |
| 3  | SET_EXPONENT (X, I)                                   | Set exponent part of a number                          |
| 4  | SPACING (X)                                           | Absolute spacing of model numbers near given number    |
| 5  |                                                       |                                                        |
| 6  | <b>13.10.13 Vector and Matrix Multiply Functions.</b> |                                                        |
| 7  | DOT_PRODUCT (VECTOR_A,<br>VECTOR_B)                   | Dot product of two rank-one arrays                     |
| 8  |                                                       |                                                        |
| 9  | MATMUL (MATRIX_A,<br>MATRIX_B)                        | Matrix multiplication                                  |
| 10 |                                                       |                                                        |
| 11 | <b>13.10.14 Array Reduction Functions.</b>            |                                                        |
| 12 | ALL (MASK, DIM)                                       | True if all values are true                            |
| 13 | Optional DIM                                          |                                                        |
| 14 | ANY (MASK, DIM)                                       | True if any value is true                              |
| 15 | Optional DIM                                          |                                                        |
| 16 | COUNT (MASK, DIM)                                     | Number of true elements in an array                    |
| 17 | Optional DIM                                          |                                                        |
| 18 | MAXVAL (ARRAY, DIM, MASK)                             | Maximum value in an array                              |
| 19 | Optional DIM, MASK                                    |                                                        |
| 20 | MINVAL (ARRAY, DIM, MASK)                             | Minimum value in an array                              |
| 21 | Optional DIM, MASK                                    |                                                        |
| 22 | PRODUCT (ARRAY, DIM, MASK)                            | Product of array elements                              |
| 23 | Optional DIM, MASK                                    |                                                        |
| 24 | SUM (ARRAY, DIM, MASK)                                | Sum of array elements                                  |
| 25 | Optional DIM, MASK                                    |                                                        |
| 26 | <b>13.10.15 Array Inquiry Functions.</b>              |                                                        |
| 27 | ALLOCATED (ARRAY)                                     | Array allocation status                                |
| 28 | LBOUND (ARRAY, DIM)                                   | Lower dimension bounds of an array                     |
| 29 | Optional DIM                                          |                                                        |
| 30 | SHAPE (SOURCE)                                        | Shape of an array or scalar                            |
| 31 | SIZE (ARRAY, DIM)                                     | Total number of elements in an array                   |
| 32 | Optional DIM                                          |                                                        |
| 33 | UBOUND (ARRAY, DIM)                                   | Upper dimension bounds of an array                     |
| 34 | Optional DIM                                          |                                                        |
| 35 | <b>13.10.16 Array Construction Functions.</b>         |                                                        |
| 36 | MERGE (TSOURCE,<br>FSOURCE, MASK)                     | Merge under mask                                       |
| 37 |                                                       |                                                        |
| 38 | PACK (ARRAY, MASK, VECTOR)                            | Pack an array into an array of rank one under a mask   |
| 39 | Optional VECTOR                                       |                                                        |
| 40 | SPREAD (SOURCE, DIM,<br>NCOPIES)                      | Replicates array by adding a dimension                 |
| 41 |                                                       |                                                        |
| 42 | UNPACK (VECTOR, MASK,<br>FIELD)                       | Unpack an array of rank one into an array under a mask |
| 43 |                                                       |                                                        |
| 44 | <b>13.10.17 Array Reshape Function.</b>               |                                                        |
| 45 | RESHAPE (SOURCE, SHAPE,<br>PAD, ORDER)                | Reshape an array                                       |
| 46 | Optional PAD, ORDER                                   |                                                        |
| 47 |                                                       |                                                        |

1 **13.10.18 Array Manipulation Functions.**

|   |                            |                                   |
|---|----------------------------|-----------------------------------|
| 2 | CSHIFT (ARRAY, SHIFT, DIM) | Circular shift                    |
| 3 | Optional DIM               |                                   |
| 4 | EOSHIFT (ARRAY, SHIFT,     | End-off shift                     |
| 5 | BOUNDARY, DIM)             |                                   |
| 6 | Optional BOUNDARY, DIM     |                                   |
| 7 | TRANSPOSE (MATRIX)         | Transpose of an array of rank two |

8 **13.10.19 Array Location Functions.**

|    |                      |                                         |
|----|----------------------|-----------------------------------------|
| 9  | MAXLOC (ARRAY, MASK) | Location of a maximum value in an array |
| 10 | Optional MASK        |                                         |
| 11 | MINLOC (ARRAY, MASK) | Location of a minimum value in an array |
| 12 | Optional MASK        |                                         |

13 **13.10.20 Pointer Association Status Inquiry Function.**

|    |                              |                                  |
|----|------------------------------|----------------------------------|
| 14 | ASSOCIATED (POINTER, TARGET) | Association status or comparison |
| 15 | Optional TARGET              |                                  |

16 **13.11 Intrinsic Subroutines.**

|    |                              |                                         |
|----|------------------------------|-----------------------------------------|
| 17 | DATE_AND_TIME (DATE, TIME,   | Obtain date and time                    |
| 18 | ZONE, VALUES)                |                                         |
| 19 | Optional DATE, TIME,         |                                         |
| 20 | ZONE, VALUES                 |                                         |
| 21 | MVBITS (FROM, FROMPOS,       | Copies bits from one integer to another |
| 22 | LEN, TO, TOPOS)              |                                         |
| 23 | RANDOM_NUMBER (HARVEST)      | Returns pseudorandom number             |
| 24 | RANDOM_SEED (SIZE, PUT, GET) | Initializes or restarts the             |
| 25 | Optional SIZE, PUT, GET      | pseudorandom number generator           |
| 26 | SYSTEM_CLOCK (COUNT,         | Obtain data from the system clock       |
| 27 | COUNT_RATE, COUNT_MAX)       |                                         |
| 28 | Optional COUNT, COUNT_RATE,  |                                         |
| 29 | COUNT_MAX                    |                                         |

30 **13.12 Specific Names for Intrinsic Functions.**

| 31 | <i>Specific Name</i>   | <i>Generic Name</i> | <i>Argument Type</i> |
|----|------------------------|---------------------|----------------------|
| 32 | ABS (A)                | ABS (A)             | default real         |
| 33 | ACOS (X)               | ACOS (X)            | default real         |
| 34 | AIMAG (Z)              | AIMAG (Z)           | default complex      |
| 35 | AINT (A)               | AINT (A)            | default real         |
| 36 | ALOG (X)               | LOG (X)             | default real         |
| 37 | ALOG10 (X)             | LOG10 (X)           | default real         |
| 38 | • AMAX0 (A1,A2,A3,...) | REAL (MAX (A1,      | default integer      |
| 39 | Optional A3,...        | A2,A3,...))         |                      |
| 40 |                        | Optional A3,...     |                      |
| 41 | • AMAX1 (A1,A2,A3,...) | MAX (A1,            | default real         |
| 42 | Optional A3,...        | A2,A3,...)          |                      |
| 43 |                        | Optional A3,...     |                      |
| 44 | • AMIN0 (A1,A2,A3,...) | REAL (MIN (A1,      | default integer      |
| 45 | Optional A3,...        | A2,A3,...))         |                      |
| 46 |                        | Optional A3,...     |                      |

## Intrinsic Procedures

|    |   |                      |                    |                       |
|----|---|----------------------|--------------------|-----------------------|
| 1  | • | AMIN1 (A1,A2,A3,...) | MIN (A1,           | default real          |
| 2  |   | Optional A3,...      | A2,A3,...)         |                       |
| 3  |   |                      | Optional A3,...    |                       |
| 4  |   | AMOD (A,P)           | MOD (A,P)          | default real          |
| 5  |   | ANINT (A)            | ANINT (A)          | default real          |
| 6  |   | ASIN (X)             | ASIN (X)           | default real          |
| 7  |   | ATAN (X)             | ATAN (X)           | default real          |
| 8  |   | ATAN2 (Y,X)          | ATAN2 (Y,X)        | default real          |
| 9  |   | CABS (A)             | ABS (A)            | default complex       |
| 10 |   | CCOS (X)             | COS (X)            | default complex       |
| 11 |   | CEXP (X)             | EXP (X)            | default complex       |
| 12 | • | CHAR (I)             | CHAR (I)           | default integer       |
| 13 |   | CLOG (X)             | LOG (X)            | default complex       |
| 14 |   | CONJG (Z)            | CONJG (Z)          | default complex       |
| 15 |   | COS (X)              | COS (X)            | default real          |
| 16 |   | COSH (X)             | COSH (X)           | default real          |
| 17 |   | CSIN (X)             | SIN (X)            | default complex       |
| 18 |   | CSQRT (X)            | SQRT (X)           | default complex       |
| 19 |   | DABS (A)             | ABS (A)            | double precision real |
| 20 |   | DACOS (X)            | ACOS (X)           | double precision real |
| 21 |   | DASIN (X)            | ASIN (X)           | double precision real |
| 22 |   | DATAN (X)            | ATAN (X)           | double precision real |
| 23 |   | DATAN2 (Y,X)         | ATAN2 (Y,X)        | double precision real |
| 24 |   | DCOS (X)             | COS (X)            | double precision real |
| 25 |   | DCOSH (X)            | COSH (X)           | double precision real |
| 26 |   | DDIM (X,Y)           | DIM (X,Y)          | double precision real |
| 27 |   | DEXP (X)             | EXP (X)            | double precision real |
| 28 |   | DIM (X,Y)            | DIM (X,Y)          | default real          |
| 29 |   | DINT (A)             | AINT (A)           | double precision real |
| 30 |   | DLOG (X)             | LOG (X)            | double precision real |
| 31 |   | DLOG10 (X)           | LOG10 (X)          | double precision real |
| 32 | • | DMAX1 (A1,A2,A3,...) | MAX (A1,A2,A3,...) | double precision real |
| 33 |   | Optional A3,...      | Optional A3,...    |                       |
| 34 | • | DMIN1 (A1,A2,A3,...) | MIN (A1,A2,A3,...) | double precision real |
| 35 |   | Optional A3,...      | Optional A3,...    |                       |
| 36 |   | DMOD (A,P)           | MOD (A,P)          | double precision real |
| 37 |   | DNINT (A)            | ANINT (A)          | double precision real |
| 38 |   | DPROD (X,Y)          | DPROD (X,Y)        | default real          |
| 39 |   | DSIGN (A,B)          | SIGN (A,B)         | double precision real |
| 40 |   | DSIN (X)             | SIN (X)            | double precision real |
| 41 |   | DSINH (X)            | SINH (X)           | double precision real |
| 42 |   | DSQRT (X)            | SQRT (X)           | double precision real |
| 43 |   | DTAN (X)             | TAN (X)            | double precision real |
| 44 |   | DTANH (X)            | TANH (X)           | double precision real |
| 45 |   | EXP (X)              | EXP (X)            | default real          |
| 46 | • | FLOAT (A)            | REAL (A)           | default integer       |
| 47 |   | IABS (A)             | ABS (A)            | default integer       |
| 48 | • | ICHAR (C)            | ICHAR (C)          | default character     |
| 49 |   | IDIM (X,Y)           | DIM (X,Y)          | default integer       |
| 50 | • | IDINT (A)            | INT (A)            | double precision real |
| 51 |   | IDNINT (A)           | NINT (A)           | double precision real |
| 52 | • | IFIX (A)             | INT (A)            | default real          |
| 53 |   | INDEX (STRING,       | INDEX (STRING,     | default character     |
| 54 |   | SUBSTRING)           | SUBSTRING)         |                       |
| 55 | • | INT (A)              | INT (A)            | default real          |
| 56 |   | ISIGN (A,B)          | SIGN (A,B)         | default integer       |
| 57 |   | LEN (STRING)         | LEN (STRING)       | default character     |

|    |   |                             |                             |                       |
|----|---|-----------------------------|-----------------------------|-----------------------|
| 1  | • | LGE (STRING_A,<br>STRING_B) | LGE (STRING_A,<br>STRING_B) | default character     |
| 2  |   |                             |                             |                       |
| 3  | • | LGT (STRING_A,<br>STRING_B) | LGT (STRING_A,<br>STRING_B) | default character     |
| 4  |   |                             |                             |                       |
| 5  | • | LLE (STRING_A,<br>STRING_B) | LLE (STRING_A,<br>STRING_B) | default character     |
| 6  |   |                             |                             |                       |
| 7  | • | LLT (STRING_A,<br>STRING_B) | LLT (STRING_A,<br>STRING_B) | default character     |
| 8  |   |                             |                             |                       |
| 9  | • | MAX0 (A1,A2,A3,...)         | MAX (A1,A2,A3,...)          | default integer       |
| 10 |   | Optional A3,...             | Optional A3,...             |                       |
| 11 | • | MAX1 (A1,A2,A3,...)         | INT (MAX (A1,A2,A3,...))    | default real          |
| 12 |   | Optional A3,...             | Optional A3,...             |                       |
| 13 | • | MIN0 (A1,A2,A3,...)         | MIN (A1,A2,A3,...)          | default integer       |
| 14 |   | Optional A3,...             | Optional A3,...             |                       |
| 15 | • | MIN1 (A1,A2,A3,...)         | INT (MIN (A1,A2,A3,...))    | default real          |
| 16 |   | Optional A3,...             | Optional A3,...             |                       |
| 17 |   | MOD (A,P)                   | MOD (A,P)                   | default integer       |
| 18 |   | NINT (A)                    | NINT (A)                    | default real          |
| 19 | • | REAL (A)                    | REAL (A)                    | default integer       |
| 20 |   | SIGN (A,B)                  | SIGN (A,B)                  | default real          |
| 21 |   | SIN (X)                     | SIN (X)                     | default real          |
| 22 |   | SINH (X)                    | SINH (X)                    | default real          |
| 23 | • | SNGL (A)                    | REAL (A)                    | double precision real |
| 24 |   | SQRT (X)                    | SQRT (X)                    | default real          |
| 25 |   | TAN (X)                     | TAN (X)                     | default real          |
| 26 |   | TANH (X)                    | TANH (X)                    | default real          |

27 • These specific intrinsic function names must not be used as an actual argument.

28 **13.13 Specifications of the Intrinsic Procedures.** This section contains detailed specifica-  
29 tions of the generic intrinsic procedures in alphabetical order.

30 **13.13.1 ABS (A).**

31 **Description.** Absolute value.

32 **Class.** Elemental function.

33 **Argument.** A must be of type integer, real, or complex.

34 **Result Type and Type Parameter.** The same as A except that if A is complex, the result is  
35 real.

36 **Result Value.** If A is of type integer or real, the value of the result is |A|; if A is complex  
37 with value (x,y), the result is equal to a processor-dependent approximation to  $\sqrt{x^2+y^2}$ .

38 **Example.** ABS ((3.0, 4.0)) has the value 5.0 (approximately).

39 **13.13.2 ACHAR (I).**

40 **Description.** Returns the character in a specified position of the ASCII collating sequence. It  
41 is the inverse of the IACHAR function.

42 **Class.** Elemental function.

43 **Argument.** I must be of type integer.

44 **Result Type and Type Parameter.** Character of length one with kind type parameter value  
45 KIND ('A').

1       **Result Value.** If I has a value in the range  $0 \leq I \leq 127$ , the result is the character in position I of  
2       the ASCII collating sequence, provided the processor is capable of representing that character;  
3       otherwise, the result is processor dependent. If the processor is not capable of representing  
4       both upper and lower case letters and I corresponds to a letter in a case that the processor  
5       is not capable of representing, the result is the letter in the case that the processor is capable  
6       of representing. ACHAR (IACHAR (C)) must have the value C for any character C capable  
7       of representation in the processor.

8       **Example.** ACHAR (88) has the value 'X'.

9       **13.13.3 ACOS (X).**

10       **Description.** Arccosine (inverse cosine) function.

11       **Class.** Elemental function.

12       **Argument.** X must be of type real with a value that satisfies the inequality  $|X| \leq 1$ .

13       **Result Type and Type Parameter.** Same as X.

14       **Result Value.** The result has a value equal to a processor-dependent approximation to  
15        $\arccos(X)$ , expressed in radians. It lies in the range  $0 \leq \text{ACOS}(X) \leq \pi$ .

16       **Example.** ACOS (0.54030231) has the value 1.0 (approximately).

17       **13.13.4 ADJUSTL (STRING).**

18       **Description.** Adjust to the left, removing leading blanks and inserting trailing blanks.

19       **Class.** Elemental function.

20       **Argument.** STRING must be of type character.

21       **Result Type.** Character of the same length and kind type parameter as STRING.

22       **Result Value.** The value of the result is the same as STRING except that any leading blanks  
23       have been deleted and the same number of trailing blanks have been inserted.

24       **Example.** ADJUSTL (' WORD') has the value 'WORD '.

25       **13.13.5 ADJUSTR (STRING).**

26       **Description.** Adjust to the right, removing trailing blanks and inserting leading blanks.

27       **Class.** Elemental function.

28       **Argument.** STRING must be of type character.

29       **Result Type.** Character of the same length and kind type parameter as STRING.

30       **Result Value.** The value of the result is the same as STRING except that any trailing blanks  
31       have been deleted and the same number of leading blanks have been inserted.

32       **Example.** ADJUSTR ('WORD ') has the value ' WORD'.

33       **13.13.6 AIMAG (Z).**

34       **Description.** Imaginary part of a complex number.

35       **Class.** Elemental function.

36       **Argument.** Z must be of type complex.

37       **Result Type and Type Parameter.** Real with the same type parameter as Z.

38       **Result Value.** If Z has the value  $(x, y)$ , the result has value  $y$ .

39       **Example.** AIMAG ((2.0, 3.0)) has the value 3.0.



1 **13.13.7 AINT (A, KIND).**2 **Optional Argument.** KIND3 **Description.** Truncation to a whole number.4 **Class.** Elemental function.5 **Arguments.**

6 A must be of type real.

7 KIND (optional) must be a scalar integer initialization expression.

8 **Result Type and Type Parameter.** The result is of type real. If KIND is present, the kind  
9 type parameter is that specified by KIND; otherwise, the kind type parameter is that of A.10 **Result Value.** If  $|A| < 1$ , AINT (A) has the value 0; if  $|A| \geq 1$ , AINT (A) has a value equal to  
11 the largest integer that does not exceed the magnitude of A and whose sign is the same as the  
12 sign of A. *integer whose magnitude is the*13 **Examples.** AINT (2.783) has the value 2.0. AINT (-2.783) has the value -2.0.14 **13.13.8 ALL (MASK, DIM).**15 **Optional Argument.** DIM16 **Description.** Determine whether all values are true in MASK along dimension DIM.17 **Class.** Transformational function.18 **Arguments.**

19 MASK must be of type logical. It must not be scalar.

20 DIM (optional) must be scalar and of type integer with value in the range  $1 \leq \text{DIM} \leq n$ ,  
21 where  $n$  is the rank of MASK.22 **Result Type, Type Parameter, and Shape.** The result is of type logical with the same kind  
23 type parameter as MASK. It is scalar if DIM is absent or MASK has rank one; otherwise, the  
24 result is an array of rank  $n-1$  and of shape  $(d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n)$  where  $(d_1, d_2, \dots,$   
25  $d_n)$  is the shape of MASK.26 **Result Value.**27 **Case (i):** The result of ALL (MASK) has the value true if all elements of MASK are true or  
28 if MASK has size zero, and the result has value false if any element of MASK is  
29 false.30 **Case (ii):** If MASK has rank one, ALL (MASK, DIM) has a value equal to that of ALL  
31 (MASK). Otherwise, the value of element  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$  of ALL  
32 (MASK, DIM) is equal to ALL (MASK  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$ ).33 **Examples.**34 **Case (i):** The value of ALL ((/ .TRUE., .FALSE., .TRUE. /)) is false.35 **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,  
36 DIM = 1) is [true, false, false] and ALL (B .NE. C, DIM = 2) is [false, false].37 **13.13.9 ALLOCATED (ARRAY).**38 **Description.** Indicate whether or not an allocatable array is currently allocated.39 **Class.** Inquiry function.40 **Argument.** ARRAY must be an allocatable array.

## Intrinsic Procedures

- 1       **Result Type, Type Parameter, and Shape.** Default logical scalar.
- 2       **Result Value.** The result has the value true if ARRAY is currently allocated and has the
- 3       value false if ARRAY is not currently allocated. The result is undefined if the allocation sta-
- 4       tus (14.8) of the array is undefined.
- 5    **13.13.10 ANINT (A, KIND).**
- 6       **Optional Argument.** KIND
- 7       **Description.** Nearest whole number.
- 8       **Class.** Elemental function.
- 9       **Arguments.**
- 10      A                    must be of type real.
- 11      KIND (optional)    must be a scalar integer initialization expression.
- 12      **Result Type and Type Parameter.** The result is of type real. If KIND is present, the kind
- 13      type parameter is that specified by KIND; otherwise, the kind type parameter is that of A.
- 14      **Result Value.** If  $A > 0$ , ANINT (A) has the value AINT (A + 0.5); if  $A \leq 0$ , ANINT (A) has the
- 15      value AINT (A - 0.5).
- 16      **Examples.** ANINT (2.783) has the value 3.0. ANINT (-2.783) has the value -3.0.

### 17    **13.13.11 ANY (MASK, DIM).**

- 18      **Optional Argument.** DIM
- 19      **Description.** Determine whether any value is true in MASK along dimension DIM.
- 20      **Class.** Transformational function.
- 21      **Arguments.**
- 22      MASK                must be of type logical. It must not be scalar.
- 23      DIM (optional)    must be scalar and of type integer with a value in the range  $1 \leq \text{DIM} \leq n$ ,
- 24      where  $n$  is the rank of MASK.
- 25      **Result Type, Type Parameter, and Shape.** The result is of type logical with the same kind
- 26      type parameter as MASK. It is scalar if DIM is absent or MASK has rank one; otherwise, the
- 27      result is an array of rank  $n-1$  and of shape  $(d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n)$  where  $(d_1, d_2, \dots,$
- 28       $d_n)$  is the shape of MASK.
- 29      **Result Value.**
- 30      *Case (i):*        The result of ANY (MASK) has the value true if any element of MASK is true
- 31      and has the value false if no elements are true or if MASK has size zero.
- 32      *Case (ii):*     If MASK has rank one, ANY (MASK, DIM) has a value equal to that of ANY
- 33      (MASK). Otherwise, the value of element  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$  of ANY
- 34      (MASK, DIM) is equal to ANY (MASK  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$ ).
- 35      **Examples.**
- 36      *Case (i):*        The value of ANY ((/ .TRUE., .FALSE., .TRUE. /)) is true.
- 37      *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
- 38      [true, false, true] and ANY (B .NE. C, DIM = 2) is [true, true].

1 **13.13.12 ASIN (X).**

2 **Description.** Arcsine (inverse sine) function.

3 **Class.** Elemental function.

4 **Argument.** X must be of type real. Its value must satisfy the inequality  $|X| \leq 1$ .

5 **Result Type and Type Parameter.** Same as X.

6 **Result Value.** The result has a value equal to a processor-dependent approximation to  $\arcsin(X)$ , expressed in radians. It lies in the range  $-\pi/2 \leq \text{ASIN}(X) \leq \pi/2$ .

8 **Example.** ASIN (0.84147098) has the value 1.0 (approximately).

9 **13.13.13 ASSOCIATED (POINTER, TARGET).**

10 **Optional Argument.** TARGET

11 **Description.** Returns the association status of its pointer argument or indicates the pointer is associated with the target.

13 **Class.** Inquiry function.

14 **Arguments.**

15 **POINTER** must be a pointer and may be of any type. Its pointer association status must not be undefined.

17 **TARGET (optional)** must be a pointer or target. *How does one discover that a pointer is undefined?* If it is a pointer, its pointer association status must not be undefined.

19 **Result Type.** The result is of type default logical.

20 **Result Value.**

21 **Case (i):** If TARGET is absent, the result is true if POINTER is currently associated with a target and false if it is not.

23 **Case (ii):** If TARGET is present and is a target, the result is true if POINTER is currently associated with TARGET and false if it is not.

25 **Case (iii):** If TARGET is present and is a pointer, the result is true if both POINTER and TARGET are currently associated with the same target, and is false otherwise. If either POINTER or TARGET is disassociated, the result is false.

28 **Examples.** ASSOCIATED (CURRENT, HEAD) is true if CURRENT points to the target HEAD. After the execution of

30 `A_PART => A (:N)`

31 ASSOCIATED (A\_PART, A) is true if N is equal to UBOUND (A, DIM = 1). After the execution of

33 `NULLIFY (CUR); NULLIFY (TOP)`

34 ASSOCIATED (CUR, TOP) is false.

35 **13.13.14 ATAN (X).**

36 **Description.** Arctangent (inverse tangent) function.

37 **Class.** Elemental function.

38 **Argument.** X must be of type real.

39 **Result Type and Type Parameter.** Same as X.

40 **Result Value.** The result has a value equal to a processor-dependent approximation to  $\arctan(X)$ , expressed in radians, that lies in the range  $-\pi/2 \leq \text{ATAN}(X) \leq \pi/2$ .

1        **Example.** ATAN (1.5574077) has the value 1.0 (approximately).

2        **13.13.15 ATAN2 (Y, X).**

3        **Description.** Arctangent (inverse tangent) function. The result is the principal value of the  
4        argument of the nonzero complex number (X, Y).

5        **Class.** Elemental function.

6        **Arguments.**

7        Y                            must be of type real.

8        X                            must be of the same type and kind type parameter as Y. If Y has the  
9        value zero, X must not have the value zero.

10       **Result Type and Type Parameter.** Same as X.

11       **Result Value.** The result has a value equal to a processor-dependent approximation to the  
12       principal value of the argument of the complex number (X, Y), expressed in radians. It lies in  
13       the range  $-\pi < \text{ATAN2}(Y, X) \leq \pi$  and is equal to a processor-dependent approximation to a  
14       value of  $\arctan(Y/X)$  if  $X \neq 0$ . If  $Y > 0$ , the result is positive. If  $Y = 0$ , the result is zero if  $X > 0$   
15       and the result is  $\pi$  if  $X < 0$ . If  $Y < 0$ , the result is negative. If  $X = 0$ , the absolute value of the  
16       result is  $\pi/2$ .

17       **Examples.** ATAN2 (1.5574077, 1.0) has the value 1.0 (approximately). If Y has the value

18        $\begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix}$  and X has the value  $\begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}$ , the value of ATAN2 (Y, X) is approximately

19        $\begin{bmatrix} \frac{3\pi}{4} & \frac{\pi}{4} \\ -\frac{3\pi}{4} & -\frac{\pi}{4} \end{bmatrix}$ .

20       **13.13.16 BIT\_SIZE (I).**

21       **Description.** Returns the number of bits *s* defined by the model of 13.5.7.

22       **Class.** Inquiry function.

23       **Argument.** I must be of type integer.

24       **Result Type, Type Parameter, and Shape.** Scalar integer with the same kind type parameter  
25       as I.

26       **Result Value.** The result has the value of the number of bits *s* in the model integer defined  
27       for bit manipulation contexts in 13.5.7.

28       **Example.** BIT\_SIZE (1) has the value 32 if *s* in the model is 32.

29       **13.13.17 BTEST (I, POS).**

30       **Description.** Tests a bit of an integer value.

31       **Class.** Elemental function.

32       **Arguments.**

33       I                            must be of type integer.

34       POS                        must be of type integer. It must be nonnegative and be less than  
35       BIT\_SIZE (I).

36       **Result Type.** The result is of type default logical.

37       **Result Value.** The result has the value true if bit POS of I has the value 1 and has the value  
38       false if bit POS of I has the value 0. The model for the interpretation of an integer value as a

1 sequence of bits is in 13.5.7.

2 **Examples.** BTEST (8, 3) has the value true. If A has the value  $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ , the value of  
 3 BTEST (A, 2) is  $\begin{bmatrix} \text{false} & \text{false} \\ \text{false} & \text{true} \end{bmatrix}$  and the value of BTEST (2, A) is  $\begin{bmatrix} \text{true} & \text{false} \\ \text{false} & \text{false} \end{bmatrix}$

#### 4 13.13.18 CEILING (A).

5 **Description.** Returns the least integer greater than or equal to its argument.

6 **Class.** Elemental function.

7 **Argument.** A must be of type real.

8 **Result Type and Type Parameter.** Default integer.

9 **Result Value.** The result has a value equal to the least integer greater than or equal to A.  
 10 The result is undefined if the processor cannot represent this value in the default integer  
 11 type.

12 **Example.** CEILING (3.7) has the value 4. CEILING (-3.7) has the value -3.

#### 13 13.13.19 CHAR (I, KIND).

14 **Optional Argument.** KIND

15 **Description.** Returns the character in a given position of the processor collating sequence  
 16 associated with the specified kind type parameter. It is the inverse of the function ICHAR.

17 **Class.** Elemental function.

18 **Arguments.**

19 I must be of type integer with a value in the range  $0 \leq I \leq n-1$ , where  $n$  is  
 20 the number of characters in the collating sequence.

21 KIND (optional) must be a scalar integer initialization expression.

22 **Result Type and Type Parameters.** Character of length one. If KIND is present, the kind  
 23 type parameter is that specified by KIND; otherwise, the kind type parameter is that of  
 24 default character type.

25 **Result Value.** The result is the character in position I of the collating sequence associated  
 26 with the specified kind type parameter. ICHAR (CHAR (I, KIND (C))) must have the value I  
 27 for  $0 \leq I \leq n-1$  and CHAR (ICHAR (C), KIND (C)) must have the value C for any character C  
 28 capable of representation in the processor.

29 **Example.** CHAR (88) has the value 'X' on a processor using the ASCII collating sequence.

#### 30 13.13.20 CMPLX (X, Y, KIND).

31 **Optional Arguments.** Y, KIND

32 **Description.** Convert to complex type.

33 **Class.** Elemental function.

34 **Arguments.**

35 X must be of type integer, real, or complex.

36 Y (optional) must be of type integer or real. It must not be present if X is of type  
 37 complex.

38 KIND (optional) must be a scalar integer initialization expression,

## Intrinsic Procedures

- 1      **Result Type and Type Parameter.** The result is of type complex. If KIND is present, the type  
2      parameter is that specified by KIND; otherwise, the type parameter is that of default real  
3      type.
- 4      **Result Value.** If Y is absent and X is not complex, it is as if Y were present with the value  
5      zero. If Y is absent and X is complex, it is as if Y were present with the value AIMAG (X).  
6      Cmplx (X, Y, KIND) has the complex value whose real part is REAL (X, KIND) and whose  
7      imaginary part is REAL (Y, KIND).
- 8      **Example.** Cmplx (-3) has the value (-3.0, 0.0).

### 9      13.13.21 CONJG (Z).

- 10     **Description.** Conjugate of a complex number.
- 11     **Class.** Elemental function.
- 12     **Argument.** Z must be of type complex.
- 13     **Result Type and Type Parameter.** Same as Z.
- 14     **Result Value.** If Z has the value  $(x, y)$ , the result has the value  $(x, -y)$ .
- 15     **Example.** CONJG ((2.0, 3.0)) has the value (2.0, -3.0).

### 16     13.13.22 COS (X).

- 17     **Description.** Cosine function.
- 18     **Class.** Elemental function.
- 19     **Argument.** X must be of type real or complex.
- 20     **Result Type and Type Parameter.** Same as X.
- 21     **Result Value.** The result has a value equal to a processor-dependent approximation to  
22      $\cos(X)$ . If X is of type real, it is regarded as a value in radians. If X is of type complex, its real  
23     part is regarded as a value in radians.
- 24     **Example.** COS (1.0) has the value 0.54030231 (approximately).

### 25     13.13.23 COSH (X).

- 26     **Description.** Hyperbolic cosine function.
- 27     **Class.** Elemental function.
- 28     **Argument.** X must be of type real.
- 29     **Result Type and Type Parameter.** Same as X.
- 30     **Result Value.** The result has a value equal to a processor-dependent approximation to  
31      $\cosh(X)$ .
- 32     **Example.** COSH (1.0) has the value 1.5430806 (approximately).

### 33     13.13.24 COUNT (MASK, DIM).

- 34     **Optional Argument.** DIM
- 35     **Description.** Count the number of true elements of MASK along dimension DIM.
- 36     **Class.** Transformational function.
- 37     **Arguments.**
- 38     MASK                    must be of type logical. It must not be scalar.
- 39     DIM (optional)         must be scalar and of type integer with a value in the range  $1 \leq \text{DIM} \leq n$ ,  
40     where  $n$  is the rank of MASK.

1 **Result Type, Type Parameter, and Shape.** The result is of type default integer. It is scalar if  
 2 DIM is absent or MASK has rank one; otherwise, the result is an array of rank  $n-1$  and of  
 3 shape  $(d_1, d_2, \dots, d_{DIM-1}, d_{DIM+1}, \dots, d_n)$  where  $(d_1, d_2, \dots, d_n)$  is the shape of MASK.

4 **Result Value.**

5 *Case (i):* The result of COUNT (MASK) has a value equal to the number of true elements  
 6 of MASK or has the value zero if MASK has size zero.

7 *Case (ii):* If MASK has rank one, COUNT (MASK, DIM) has a value equal to that of  
 8 COUNT (MASK). Otherwise, the value of element  $(s_1, s_2, \dots, s_{DIM-1}, s_{DIM+1}, \dots,$   
 9  $s_n)$  of COUNT (MASK, DIM) is equal to COUNT (MASK  $(s_1, s_2, \dots, s_{DIM-1},$   
 10  $s_{DIM+1}, \dots, s_n)$ ).

11 **Examples.**

12 *Case (i):* The value of COUNT ((/ .TRUE., .FALSE., .TRUE. /)) is 2.

13 *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,  
 14 DIM = 1) is [2, 0, 1] and COUNT (B.NE.C, DIM = 2) is [1, 2].

15 **13.13.25 CSHIFT (ARRAY, SHIFT, DIM).**

16 **Optional Argument.** DIM

17 **Description.** Perform a circular shift on an array expression of rank one or perform circular  
 18 shifts on all the complete rank one sections along a given dimension of an array expression of  
 19 rank two or greater. Elements shifted out at one end of a section are shifted in at the other  
 20 end. Different sections may be shifted by different amounts and in different directions.

21 **Class.** Transformational function.

22 **Arguments.**

23 ARRAY may be of any type. It must not be scalar.

24 SHIFT must be of type integer and must be scalar if ARRAY has rank one; oth-  
 25 erwise, it must be scalar or of rank  $n-1$  and of shape  $(d_1, d_2, \dots, d_{DIM-1},$   
 26  $d_{DIM+1}, \dots, d_n)$  where  $(d_1, d_2, \dots, d_n)$  is the shape of ARRAY.

27 DIM (optional) must be a scalar and of type integer with a value in the range  
 28  $1 \leq DIM \leq n$ , where  $n$  is the rank of ARRAY. If DIM is omitted, it is as if  
 29 it were present with the value 1.

30 **Result Type, Type Parameter, and Shape.** The result is of the type and type parameters of  
 31 ARRAY, and has the shape of ARRAY.

32 **Result Value.**

33 *Case (i):* If ARRAY has rank one, element  $i$  of the result is ARRAY  $(1 + \text{MODULO}(i +$   
 34  $\text{SHIFT} - 1, \text{SIZE}(\text{ARRAY})))$ .

35 *Case (ii):* If ARRAY has rank greater than one, section  $(s_1, s_2, \dots, s_{DIM-1}, s_{DIM+1}, \dots, s_n)$  of  
 36 the result has a value equal to CSHIFT (ARRAY  $(s_1, s_2, \dots, s_{DIM-1}, s_{DIM+1}, \dots,$   
 37  $s_n), 1, sh)$ , where  $sh$  is SHIFT or SHIFT  $(s_1, s_2, \dots, s_{DIM-1}, s_{DIM+1}, \dots, s_n)$ .

38 **Examples.**

39 *Case (i):* If V is the array [1, 2, 3, 4, 5, 6], the effect of shifting V circularly to the left by  
 40 two positions is achieved by CSHIFT (V, SHIFT = 2) which has the value  
 41 [3, 4, 5, 6, 1, 2]; CSHIFT (V, SHIFT = -2) achieves a circular shift to the right by  
 42 two positions and has the value [5, 6, 1, 2, 3, 4].

43 *Case (ii):* The rows of an array of rank two may all be shifted by the same amount or by  
 44 different amounts. If M is the array  $\begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix}$ , the value of CSHIFT (M,

1                   SHIFT = -1, DIM = 2) is  $\begin{bmatrix} C & A & B \\ F & D & E \\ I & G & H \end{bmatrix}$ , and the value of CSHIFT (M, SHIFT =

2                   (/ -1, 1, 0 /), DIM = 2) is  $\begin{bmatrix} C & A & B \\ E & F & D \\ G & H & I \end{bmatrix}$ .

3   **13.13.26 DATE\_AND\_TIME (DATE, TIME, ZONE, VALUES).**

4       **Optional Arguments.** DATE, TIME, ZONE, VALUES

5       **Description.** Returns ~~character~~ data on the real-time clock and date in a form compatible  
6       with the representations defined in ISO 8601:1988.

7       **Class.** Subroutine.

8       **Arguments.**

9       DATE (optional)    must be scalar and type default character, and must be of length at least  
10                           8 in order to contain the complete value. It is an INTENT (OUT) argu-  
11                           ment. Its leftmost 8 characters are set to a value of the form  
12                           CCYYMMDD, where CC is the century, YY the year within the century,  
13                           MM the month within the year, and DD the day within the month. If  
14                           there is no date available, they are set to blank.

15       TIME (optional)   must be scalar and type default character, and must be of length at least  
16                           10 in order to contain the complete value. It is an INTENT (OUT) argu-  
17                           ment. Its leftmost 10 characters are set to a value of the form  
18                           hhmmss.sss, where hh is the hour of the day, mm is the minutes of the  
19                           hour, and ss.sss is the seconds and milliseconds of the minute. If there is  
20                           no clock available, they are set to blank.

21       ZONE (optional)   must be scalar and type default character, and must be of length at least  
22                           5 in order to contain the complete value. It is an INTENT (OUT) argu-  
23                           ment. Its leftmost 5 characters are set to a value of the form ±hhmm,  
24                           where hh and mm are the time difference with respect to Coordinated  
25                           Universal Time (UTC) in hours and parts of an hour expressed in min-  
26                           utes, respectively. If there is no clock available, they are set to blank.

27       VALUES (optional) must be of type default integer and of rank one. It is an INTENT (OUT)  
28                           argument. Its size must be at least 8. The values returned in VALUES  
29                           are as follows:

30           VALUES (1)    the year (for example, 1990), or -HUGE (0) if there is no date available;

31           VALUES (2)    the month of the year, or -HUGE (0) if there is no date available;

32           VALUES (3)    the day of the month, or -HUGE (0) if there is no date available;

33           VALUES (4)    the time difference with respect to Coordinated Universal Time (UTC)  
34                           in minutes, or -HUGE (0) if this information is not available;

35           VALUES (5)    the hour of the day, in the range of 0 to 23, or -HUGE (0) if there is no  
36                           clock;

37           VALUES (6)    the minutes of the hour, in the range 0 to 59, or -HUGE (0) if there is no  
38                           clock;

39           VALUES (7)    the seconds of the minute, in the range 0 to 60, or -HUGE (0) if there is  
40                           no clock;

41           VALUES (8)    the milliseconds of the second, in the range 0 to 999, or -HUGE (0) if  
42                           there is no clock.



1     **Example.**

```

2           INTEGER DATE_TIME (8)
3           CHARACTER (LEN = 10) BIG_BEN (3)
4           CALL DATE_AND_TIME (BIG_BEN (1), BIG_BEN (2), &
5                               BIG_BEN (3), DATE_TIME)

```

6     if called in Geneva, Switzerland on 1985 April 12 at 15:27:35.5 would have assigned the value  
7     19850412 to BIG\_BEN (1), the value 152735.500 to BIG\_BEN (2), and the value +0100 to  
8     BIG\_BEN (3), and the following values to DATE\_TIME: 1985, 4, 12, 60, 15, 27, 35, 500.

9     Note that UTC is defined by CCIR Recommendation 460-2 (and is also known as Greenwich  
10    Mean Time).

11   **13.13.27 DBLE (A).**

12    **Description.** Convert to double precision real type.

13    **Class.** Elemental function.

14    **Argument.** A must be of type integer, real, or complex.

15    **Result Type and Type Parameter.** Double precision real.

16    **Result Value.**

17    *Case (i):*     If A is of type double precision real, DBLE (A) = A.

18    *Case (ii):*    If A is of type integer or real, the result is as much precision of the significant  
19                      part of A as a double precision real datum can contain.

20    *Case (iii):*   If A is of type complex, the result is as much precision of the significant part of  
21                      the real part of A as a double precision real datum can contain.

22    **Example.** DBLE (-3) has the value -3.0D0.

23   **13.13.28 DIGITS (X).**

24    **Description.** Returns the number of significant digits in the model representing numbers of  
25    the same type and type parameter as the argument.

26    **Class.** Inquiry function.

27    **Argument.** X must be of type integer or real. It may be scalar or array valued.

28    **Result Type, Type Parameter, and Shape.** Default integer scalar.

29    **Result Value.** The result has the value  $q$  if X is of type integer and  $p$  if X is of type real, where  
30     $q$  and  $p$  are as defined in 13.7.1 for the model representing numbers of the same type and type  
31    parameter as X.

32    **Example.** DIGITS (X) has the value 24 for real X whose model is as at the end of 13.7.1.

33   **13.13.29 DIM (X, Y).**

34    **Description.** The difference  $X-Y$  if it is positive; otherwise zero.

35    **Class.** Elemental function.

36    **Arguments.**

37    X                    must be of type integer or real.

38    Y                    must be of the same type and kind type parameter as X.

39    **Result Type and Type Parameter.** Same as X.

40    **Result Value.** The value of the result is  $X-Y$  if  $X > Y$  and zero otherwise.

1       **Example.** DIM (-3.0, 2.0) has the value 0.0.

2       **13.13.30 DOT\_PRODUCT (VECTOR\_A, VECTOR\_B).**

3       **Description.** Performs dot-product multiplication of numeric or logical vectors.

4       **Class.** Transformational function.

5       **Arguments.**

6       VECTOR\_A        must be of numeric type (integer, real, or complex) or of logical type. It  
7                            must be array valued and of rank one.

8       VECTOR\_B        must be of numeric type if VECTOR\_A is of numeric type or of type  
9                            logical if VECTOR\_A is of type logical. It must be array valued and of  
10                           rank one. It must be of the same size as VECTOR\_A.

11       **Result Type, Type Parameter, and Shape.** If the arguments are of numeric type, the type  
12                           and type parameter of the result are those of the expression VECTOR\_A \* VECTOR\_B deter-  
13                           mined by the types of the arguments according to 7.1.4. If the arguments are of type logical,  
14                           the result is of type logical with the type parameter of the expression VECTOR\_A .AND.  
15                           VECTOR\_B according to 7.1.4. The result is scalar.

16       **Result Value.**

17       Case (i):        If VECTOR\_A is of type integer or real, the result has the value SUM  
18                           (VECTOR\_A\*VECTOR\_B). If the vectors have size zero, the result has the value  
19                           zero.

20       Case (ii):       If VECTOR\_A is of type complex, the result has the value SUM (CONJG  
21                           (VECTOR\_A)\*VECTOR\_B). If the vectors have size zero, the result has the  
22                           value zero.

23       Case (iii):     If VECTOR\_A is of type logical, the result has the value ANY (VECTOR\_A  
24                           .AND. VECTOR\_B). If the vectors have size zero, the result has the value false.

25       **Example.** DOT\_PRODUCT ((/ 1, 2, 3 /), (/ 2, 3, 4 /)) has the value 20.

26       **13.13.31 DPROD (X, Y).**

27       **Description.** Double precision real product.

28       **Class.** Elemental function.

29       **Arguments.**

30       X                must be of type default real.

31       Y                must be of type default real.

32       **Result Type and Type Parameters.** Double precision real.

33       **Result Value.** The result has a value equal to a processor-dependent approximation to the  
34                           product of X and Y.

35       **Example.** DPROD (-3.0, 2.0) has the value -6.0D0.

36       **13.13.32 EOSHIFT (ARRAY, SHIFT, BOUNDARY, DIM).**

37       **Optional Arguments.** BOUNDARY, DIM

38       **Description.** Perform an end-off shift on an array expression of rank one or perform end-off  
39                           shifts on all the complete rank-one sections along a given dimension of an array expression of  
40                           rank two or greater. Elements are shifted off at one end of a section and copies of a boundary  
41                           value are shifted in at the other end. Different sections may have different boundary values  
42                           and may be shifted by different amounts and in different directions.

1 **Class.** Transformational function.  
 2 **Arguments.**  
 3 **ARRAY** may be of any type. It must not be scalar.  
 4 **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  $(d_1, d_2, \dots, d_{DIM-1}, d_{DIM+1}, \dots, d_n)$  where  $(d_1, d_2, \dots, d_n)$  is the shape of ARRAY.  
 5  
 6  
 7 **BOUNDARY** (optional)  
 8 must be of the same type and type parameters as ARRAY and must be  
 9 scalar if ARRAY has rank one; otherwise, it must be either scalar or of  
 10 rank  $n-1$  and of shape  $(d_1, d_2, \dots, d_{DIM-1}, d_{DIM+1}, \dots, d_n)$ . BOUNDARY  
 11 may be omitted for the data types in the following table and, in this  
 12 case, it is as if it were present with the scalar value shown.

| Type of ARRAY            | Value of BOUNDARY |
|--------------------------|-------------------|
| Integer                  | 0                 |
| Real                     | 0.0               |
| Complex                  | (0.0, 0.0)        |
| Logical                  | false             |
| Character ( <i>len</i> ) | <i>len</i> blanks |

20 **DIM** (optional) must be scalar and of type integer with a value in the range  $1 \leq DIM \leq n$ ,  
 21 where  $n$  is the rank of ARRAY. If DIM is omitted, it is as if it were present  
 22 with the value 1.

23 **Result Type, Type Parameter, and Shape.** The result has the type, type parameters, and  
 24 shape of ARRAY.

25 **Result Value.** Element  $(s_1, s_2, \dots, s_n)$  of the result has the value ARRAY  $(s_1, s_2, \dots, s_{DIM-1},$   
 26  $s_{DIM+sh}, s_{DIM+1}, \dots, s_n)$  where  $sh$  is SHIFT or SHIFT  $(s_1, s_2, \dots, s_{DIM-1}, s_{DIM+1}, \dots, s_n)$  provided the  
 27 inequality  $LBOUND(ARRAY, DIM) \leq s_{DIM} + sh \leq UBOUND(ARRAY, DIM)$  holds and is  
 28 otherwise BOUNDARY or BOUNDARY  $(s_1, s_2, \dots, s_{DIM-1}, s_{DIM+1}, \dots, s_n)$ .

29 **Examples.**

30 *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  
 31 positions is achieved by EOSHIFT (V, SHIFT = 3) which has the value  
 32 [4, 5, 6, 0, 0, 0]; EOSHIFT (V, SHIFT = -2, BOUNDARY = 99) achieves an end-off  
 33 shift to the right by 2 positions with the boundary value of 99 and has the value  
 34 [99, 99, 1, 2, 3, 4].

35 *Case (ii):* The rows of an array of rank two may all be shifted by the same amount or by  
 36 different amounts and the boundary elements can be the same or different. If M

37 is the array  $\begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix}$ , then the value of EOSHIFT (M, SHIFT = -1, <sup>/new line</sup>BOUND-

38 ARY = '\*', DIM = 2) is  $\begin{bmatrix} * & A & B \\ * & D & E \\ * & G & H \end{bmatrix}$ , and the value of EOSHIFT (M,

39 SHIFT = (/ -1, 1, 0 /), BOUNDARY = (/ '\*', '/', '?' /), DIM = 2) is  $\begin{bmatrix} * & A & B \\ E & F & / \\ G & H & I \end{bmatrix}$ .

Intrinsic Procedures

*It is inscriptable that XSTS feels it is not important to define this, but feels things that are poorly defined in the presence of this feature are bad ideas.*

*b-p  
or b<sup>1-p</sup>?  
OK*

1 **13.13.33 EPSILON (X).**

2 **Description.** Returns a positive model number that is almost negligible compared to unity in

3 the model representing numbers of the same type and type parameter as the argument.

4 **Class.** Inquiry function.

5 **Argument.** X must be of type real. It may be scalar or array valued.

6 **Result Type, Type Parameter, and Shape.** Scalar of the same type and type parameter as X.

7 **Result Value.** The result has the value  $b^{1-p}$  where  $b$  and  $p$  are as defined in 13.7.1 for the

8 model representing numbers of the same type and type parameter as X.

9 **Example.** EPSILON (X) has the value  $2^{-23}$  for real X whose model is as at the end of 13.7.1.

10 **13.13.34 EXP (X).**

11 **Description.** Exponential.

12 **Class.** Elemental function.

13 **Argument.** X must be of type real or complex.

14 **Result Type and Type Parameter.** Same as X.

15 **Result Value.** The result has a value equal to a processor-dependent approximation to  $e^X$ . If

16 X is of type complex, its imaginary part is regarded as a value in radians.

17 **Example.** EXP (1.0) has the value 2.7182818 (approximately).

18 **13.13.35 EXPONENT (X).**

19 **Description.** Returns the exponent part of the argument when represented as a model num-

20 ber.

21 **Class.** Elemental function.

22 **Argument.** X must be of type real.

23 **Result Type.** Default integer.

24 **Result Value.** The result has a value equal to the exponent  $e$  of the model representation

25 (13.7.1) for the value of X, provided X is nonzero and  $e$  is within the range for integers.

26 EXPONENT (X) has the value zero if X is zero. *The result is undefined if the*

27 *processor cannot represent  $e$  in the default integer type.*

28 **Examples.** EXPONENT (1.0) has the value 1 and EXPONENT (4.1) has the value 3 for reals whose model is as at the end of 13.7.1.

29 **13.13.36 FLOOR (A).**

30 **Description.** Returns the greatest integer less than or equal to its argument.

31 **Class.** Elemental function.

32 **Argument.** A must be of type real.

33 **Result Type and Type Parameter.** Default integer.

34 **Result Value.** The result has value equal to the greatest integer less than or equal to A. The

35 result is undefined if the processor cannot represent this value in the default integer type.

36 **Example.** FLOOR (3.7) has the value 3. FLOOR (-3.7) has the value -4.

37 **13.13.37 FRACTION (X).**

38 **Description.** Returns the fractional part of the model representation of the argument value.

39 **Class.** Elemental function.

- 1     **Argument.** X must be of type real.
- 2     **Result Type and Type Parameter.** Same as X.
- 3     **Result Value.** The result has the value  $X \times b^{-e}$ , where *b* and *e* are as defined in 13.7.1 for the
- 4     model representation of X. If X has the value zero, the result has the value zero.
- 5     **Example.** FRACTION (3.0) has the value 0.75 for reals whose model is as at the end of 13.7.1.

6     **13.13.38 HUGE (X).**

- 7     **Description.** Returns the largest number in the model representing numbers of the same
- 8     type and type parameter as the argument.
- 9     **Class.** Inquiry function.
- 10    **Argument.** X must be of type integer or real. It may be scalar or array valued.
- 11    **Result Type, Type Parameter, and Shape.** Scalar of the same type and type parameter as X.
- 12    **Result Value.** The result has the value  $r^q - 1$  if X is of type integer and  $(1 - b^{-p})b^{e_{max}}$  if X is of
- 13    type real, where *r*, *q*, *b*, *p*, and  $e_{max}$  are as defined in 13.7.1 for the model representing num-
- 14    bers of the same type and type parameter as X.
- 15    **Example.** HUGE (X) has the value  $(1 - 2^{-24}) \times 2^{127}$  for real X whose model is as at the end of
- 16    13.7.1.

17    **13.13.39 IACHAR (C).**

- 18    **Description.** Returns the position of a character in the ASCII collating sequence.
- 19    **Class.** Elemental function.
- 20    **Argument.** C must be of type default character and of length one.
- 21    **Result Type and Type Parameter.** Default integer.
- 22    **Result Value.** If C is in the collating sequence defined by the codes specified in ANSI X3.4-
- 23    1986 (ASCII), the result is the position of C in that sequence and satisfies the inequality
- 24     $(0 \leq \text{IACHAR}(C) \leq 127)$ . A processor-dependent value is returned if C is not in the ASCII col-
- 25    lating sequence. The results ~~must be~~ consistent with the LGE, LGT, LLE, and LLT lexical
- 26    comparison functions. For example, if LLE (C, D) is true, IACHAR (C) .LE. IACHAR (D) is
- 27    true where C and D are any two characters representable by the processor.
- 28    **Example.** IACHAR ('X') has the value 88. *are*

29    **13.13.40 IAND (I, J).**

- 30    **Description.** Performs a logical AND.
- 31    **Class.** Elemental function.
- 32    **Arguments.**
- 33    I                    must be of type integer.
- 34    J                    must be of type integer with the same kind parameter as I.
- 35    **Result Type and Type Parameter.** Same as I.
- 36    **Result Value.** The result has the value obtained by combining I and J bit-by-bit according to
- 37    the following truth table:

| I | J | IAND (I, J) |
|---|---|-------------|
| 1 | 1 | 1           |
| 1 | 0 | 0           |
| 0 | 1 | 0           |

2 The model for the interpretation of an integer value as a sequence of bits is in 13.5.7.

3 **Example.** IAND (1, 3) has the value 1.

#### 4 **13.13.41 IBCLR (I, POS).**

5 **Description.** Clears one bit to zero.

6 **Class.** Elemental function.

7 **Arguments.**

8 I must be of type integer.

9 POS must be of type integer. It must be nonnegative and less than BIT\_SIZE  
10 (I).

11 **Result Type and Type Parameter.** Same as I.

12 **Result Value.** The result has the value of the sequence of bits of I, except that bit POS of I is  
13 set to zero. The model for the interpretation of an integer value as a sequence of bits is in  
14 13.5.7.

15 **Examples.** IBCLR (14, 1) has the result 12. If V has the value [1, 2, 3, 4], the value of  
16 IBCLR (POS = V, I = 31) is [29, 27, 23, 15].

#### 17 **13.13.42 IBITS (I, POS, LEN).**

18 **Description.** Extracts a sequence of bits.

19 **Class.** Elemental function.

20 **Arguments.**

21 I must be of type integer.

22 POS must be of type integer. It must be nonnegative and POS + LEN must  
23 be less than or equal to BIT\_SIZE (I).

24 LEN must be of type integer and nonnegative.

25 **Result Type and Type Parameter.** Same as I.

26 **Result Value.** The result has the value of the sequence of LEN bits in I beginning at bit POS  
27 right-adjusted and with all other bits zero. The model for the interpretation of an integer  
28 value as a sequence of bits is in 13.5.7.

29 **Example.** IBITS (14, 1, 3) has the value 7.

#### 30 **13.13.43 IBSET (I, POS).**

31 **Description.** Sets one bit to one.

32 **Class.** Elemental function.

33 **Arguments.**

34 I must be of type integer.

35 POS must be of type integer. It must be nonnegative and less than BIT\_SIZE  
36 (I).

37 **Result Type and Type Parameter.** Same as I.

38 **Result Value.** The result has the value of the sequence of bits of I, except that bit POS of I is  
39 set to one. The model for the interpretation of an integer value as a sequence of bits is in  
40 13.5.7.

1       **Examples.** IBSET (12, 1) has the value 14. If V has the value [1, 2, 3, 4], the value of  
 2       IBSET (POS = V, I = 0) is [2, 4, 8, 16].

3       **13.13.44 ICHAR (C).**

4       **Description.** Returns the position of a character in the processor collating sequence associ-  
 5       ated with the kind type parameter of the character.

6       **Class.** Elemental function.

7       **Argument.** C must be of type character and of length one. Its value must be that of a charac-  
 8       ter capable of representation in the processor.

9       **Result Type and Type Parameter.** Default integer.

10       **Result Value.** The result is the position of C in the processor collating sequence associated  
 11       with the kind type parameter of C and is in the range  $0 \leq \text{ICHAR}(C) \leq n-1$ , where  $n$  is the  
 12       number of characters in the collating sequence. For any characters C and D capable of repre-  
 13       sentation in the processor, C .LE. D is true if and only if ICHAR (C) .LE. ICHAR (D) is true  
 14       and C.EQ. D is true if and only if ICHAR (C).EQ. ICHAR (D) is true.

15       **Example.** ICHAR ('X') has the value 88 on a processor using the ASCII collating sequence for  
 16       the default character type.

17       **13.13.45 IEOR (I, J).**

18       **Description.** Performs an exclusive OR.

19       **Class.** Elemental function.

20       **Arguments.**

21       I                       must be of type integer.

22       J                       must be of type integer with the same kind type parameter as I.

23       **Result Type and Type Parameter.** Same as I.

24       **Result Value.** The result has the value obtained by combining I and J bit-by-bit according to  
 25       the following truth table:

| I | J | IEOR (I, J) |
|---|---|-------------|
| 1 | 1 | 0           |
| 1 | 0 | 1           |
| 0 | 1 | 1           |
| 0 | 0 | 0           |

32       The model for the interpretation of an integer value as a sequence of bits is in 13.5.7.

33       **Example.** IEOR (1, 3) has the value 2.

34       **13.13.46 INDEX (STRING, SUBSTRING, BACK).**

35       **Optional Argument.** BACK

36       **Description.** Returns the starting position of a substring within a string.

37       **Class.** Elemental function.

38       **Arguments.**

39       STRING                must be of type character.

40       SUBSTRING            must be of type character with the same kind type parameter as  
 41       STRING.

## Intrinsic Procedures

1 BACK (optional) must be of type logical.  
2 **Result Type and Type Parameter.** Default integer.

3 **Result Value.**

4 *Case (i):* If BACK is absent or present with the value false, the result is the minimum  
5 value of I such that  $\text{STRING}(I : I + \text{LEN}(\text{SUBSTRING}) - 1) = \text{SUBSTRING}$  or  
6 zero if there is no such value. Zero is returned if  $\text{LEN}(\text{STRING}) < \text{LEN}(\text{SUB-}$   
7  $\text{STRING})$  and ~~one~~ <sup>zero</sup> is returned if  $\text{LEN}(\text{SUBSTRING}) = 0$ .

8 *Case (ii):* If BACK is present with the value true, the result is the maximum value of I  
9 such that  $\text{STRING}(I : I + \text{LEN}(\text{SUBSTRING}) - 1) = \text{SUBSTRING}$  or zero if  
10 there is no such value. Zero is returned if  $\text{LEN}(\text{STRING}) < \text{LEN}(\text{SUBSTRING})$   
11 and ~~LEN(STRING) = 1~~ <sup>zero</sup> is returned if  $\text{LEN}(\text{SUBSTRING}) = 0$ .

12 **Examples.** INDEX('FORTRAN', 'R') has the value 3.  
13 INDEX('FORTRAN', 'R', BACK = .TRUE.) has the value 5.

### 14 13.13.47 INT (A, KIND).

15 **Optional Argument.** KIND

16 **Description.** Convert to integer type.

17 **Class.** Elemental function.

18 **Arguments.**

19 A must be of type integer, real, or complex.

20 KIND (optional) must be a scalar integer initialization expression.

21 **Result Type and Type Parameter.** Integer. If KIND is present, the kind type parameter is  
22 that specified by KIND; otherwise, the kind type parameter is that of default integer type.

23 **Result Value.**

24 *Case (i):* If A is of type integer,  $\text{INT}(A) = A$ .

25 *Case (ii):* If A is of type real, there are two cases: if  $|A| < 1$ ,  $\text{INT}(A)$  has the value 0; if  
26  $|A| \geq 1$ ,  $\text{INT}(A)$  is the integer whose magnitude is the largest integer that does  
27 not exceed the magnitude of A and whose sign is the same as the sign of A.

28 *Case (iii):* If A is of type complex,  $\text{INT}(A)$  is the value obtained by applying the case (ii)  
29 rule to the real part of A.

30 **Example.**  $\text{INT}(-3.7)$  has the value -3. *The value is rounded if the processor cannot represent the result in the specified integer type.*

### 31 13.13.48 IOR (I, J).

32 **Description.** Performs an inclusive OR.

33 **Class.** Elemental function.

34 **Arguments.**

35 I must be of type integer.

36 J must be of type integer with the same kind type parameter as I.

37 **Result Type and Type Parameter.** Same as I.

38 **Result Value.** The result has the value obtained by combining I and J bit-by-bit according to  
39 the following truth table:

| I | J | IOR(I, J) |
|---|---|-----------|
| 1 | 1 | 1         |



|   |   |   |
|---|---|---|
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

4 The model for the interpretation of an integer value as a sequence of bits is in 13.5.7.

5 **Example.** IOR (1, 3) has the value 3.

#### 6 **13.13.49 ISHFT (I, SHIFT).**

7 **Description.** Performs a logical shift.

8 **Class.** Elemental function.

9 **Arguments.**

10 I must be of type integer.

11 SHIFT must be of type integer. The absolute value of SHIFT must be less than  
12 or equal to BIT\_SIZE (I).

13 **Result Type and Type Parameter.** Same as I.

14 **Result Value.** The result has the value obtained by shifting the bits of I by SHIFT positions.  
15 If SHIFT is positive, the shift is to the left; if SHIFT is negative, the shift is to the right; and if  
16 SHIFT is zero, no shift is performed. Bits shifted out from the left or from the right, as appro-  
17 priate, are lost. Zeros are shifted in from the opposite end. The model for the interpretation  
18 of an integer value as a sequence of bits is in 13.5.7.

19 **Example.** ISHFT (3, 1) has the result 6.

#### 20 **13.13.50 ISHFTC (I, SHIFT, SIZE).**

21 **Optional Argument.** SIZE

22 **Description.** Performs a circular shift of the rightmost bits.

23 **Class.** Elemental function.

24 **Arguments.**

25 I must be of type integer.

26 SHIFT must be of type integer. The absolute value of SHIFT must be less than  
27 or equal to SIZE.

28 SIZE (optional) must be of type integer. The value of SIZE must be positive and must  
29 not exceed BIT\_SIZE (I). If SIZE is absent, it is as if it were present with  
30 the value of BIT\_SIZE (I).

31 **Result Type and Type Parameter.** Same as I.

32 **Result Value.** The result has the value obtained by shifting the SIZE rightmost bits of I circu-  
33 larly by SHIFT positions. If SHIFT is positive, the shift is to the left; if SHIFT is negative, the  
34 shift is to the right; and if SHIFT is zero, no shift is performed. No bits are lost. The  
35 unshifted bits are unaltered. The model for the interpretation of an integer value as a  
36 sequence of bits is in 13.5.7.

37 **Example.** ISHFTC (3, 2, 3) has the value 5.

#### 38 **13.13.51 KIND (X).**

39 **Description.** Returns the value of the kind type parameter of X.

40 **Class.** Inquiry function.

41 **Argument.** X may be of any intrinsic type.

## Intrinsic Procedures

- 1        **Result Type, Type Parameter, and Shape.** Default integer scalar.  
2        **Result Value.** The result has a value equal to the kind type parameter value of X.  
3        **Example.** KIND (0.0) has the kind type parameter value of default real.

### 4    13.13.52 LBOUND (ARRAY, DIM).

- 5        **Optional Argument.** DIM  
6        **Description.** Returns all the lower bounds or a specified lower bound of an array.  
7        **Class.** Inquiry function.

#### 8        **Arguments.**

- 9        ARRAY                may be of any type. It must not be scalar. It must not be a pointer that  
10                                is disassociated or an allocatable array that is not allocated.  
11        DIM (optional)        must be scalar and of type integer with a value in the range  $1 \leq \text{DIM} \leq n$ ,  
12                                where  $n$  is the rank of ARRAY.

- 13        **Result Type, Type Parameter, and Shape.** The result is of type default integer. It is scalar if  
14        DIM is present; otherwise, the result is an array of rank one and size  $n$ , where  $n$  is the rank of  
15        ARRAY.

#### 16        **Result Value.**

- 17        *Case (i):*        LBOUND (ARRAY, DIM) has a value equal to the lower bound for subscript  
18                                DIM of ARRAY if dimension DIM of ARRAY does not have size zero and has  
19                                the value 1 if dimension DIM has size zero. For an array section or an array  
20                                expression, it has the value 1.

- 21        *Case (ii):*        LBOUND (ARRAY) has a value whose  $i$ th component is equal to LBOUND  
22                                (ARRAY,  $i$ ), for  $i = 1, 2, \dots, n$ , where  $n$  is the rank of ARRAY.

- 23        **Example.** If A is declared by the statement

24                REAL A (2:3, 7:10)

- 25        then LBOUND (A) is [2, 7] and LBOUND (A, DIM=2) is 7.

### 26    13.13.53 LEN (STRING).

- 27        **Description.** Returns the length of a character entity.  
28        **Class.** Inquiry function.  
29        **Argument.** STRING must be of type character. It may be scalar or array valued.

- 30        **Result Type, Type Parameter, and Shape.** Default integer scalar.

- 31        **Result Value.** The result has a value equal to the number of characters in STRING if it is sca-  
32        lar or in an element of STRING if it is array valued.

- 33        **Example.** If C is declared by the statement

34                CHARACTER (11) C (100)

- 35        LEN (C) has the value 11.

### 36    13.13.54 LEN\_TRIM (STRING).

- 37        **Description.** Returns the length of the character argument without counting trailing blank  
38        characters.

- 39        **Class.** Elemental function.

- 40        **Argument.** STRING must be of type character.

1 **Result Type and Type Parameter.** Default integer.

2 **Result Value.** The result has a value equal to the number of characters <sup>remaining</sup> after any trailing  
3 blanks in STRING are removed. If the argument contains no nonblank characters, the result  
4 is zero.

5 **Examples.** LEN\_TRIM (' A B ') has the value 4 and LEN\_TRIM (' ') has the value 0.

### 6 13.13.55 LGE (STRING\_A, STRING\_B).

7 **Description.** Test whether a string is lexically greater than or equal to another string, based  
8 on the ASCII collating sequence.

9 **Class.** Elemental function.

10 **Arguments.**

11 STRING\_A must be of type default character.

12 STRING\_B must be of type default character.

13 **Result Type and Type Parameters.** Default logical.

14 **Result Value.** If the strings are of unequal length, the comparison is made as if the shorter  
15 string were extended on the right with blanks to the length of the longer string. If either  
16 string contains a character not in the ASCII character set, the result is processor dependent.  
17 The result is true if the strings are equal or if STRING\_A follows STRING\_B in the ASCII col-  
18 lating sequence; otherwise, the result is false. *The result is TRUE if both strings*  
*are of zero length.*

19 **Example.** LGE ('ONE', 'TWO') has the value false.

### 20 13.13.56 LGT (STRING\_A, STRING\_B).

21 **Description.** Test whether a string is lexically greater than another string, based on the  
22 ASCII collating sequence.

23 **Class.** Elemental function.

24 **Arguments.**

25 STRING\_A must be of type default character.

26 STRING\_B must be of type default character.

27 **Result Type and Type Parameters.** Default logical.

28 **Result Value.** If the strings are of unequal length, the comparison is made as if the shorter  
29 string were extended on the right with blanks to the length of the longer string. If either  
30 string contains a character not in the ASCII character set, the result is processor dependent.  
31 The result is true if STRING\_A follows STRING\_B in the ASCII collating sequence; otherwise,  
32 the result is false. The result is false if both STRING\_A and STRING\_B are zero length.

33 **Example.** LGT ('ONE', 'TWO') has the value false.

### 34 13.13.57 LLE (STRING\_A, STRING\_B).

35 **Description.** Test whether a string is lexically less than or equal to another string, based on  
36 the ASCII collating sequence.

37 **Class.** Elemental function.

38 **Arguments.**

39 STRING\_A must be of type default character.

40 STRING\_B must be of type default character.

41 **Result Type and Type Parameters.** Default logical.

## Intrinsic Procedures

1       **Result Value.** If the strings are of unequal length, the comparison is made as if the shorter  
2       string were extended on the right with blanks to the length of the longer string. If either  
3       string contains a character not in the ASCII character set, the result is processor dependent.  
4       The result is true if the strings are equal or if `STRING_A` precedes `STRING_B` in the ASCII  
5       collating sequence; otherwise, the result is false. *The result is .TRUE. if*  
6       *both strings are of non-zero length.*  
      **Example.** `LLE ('ONE', 'TWO')` has the value true.

### 7       **13.13.58 LLT (STRING\_A, STRING\_B).**

8       **Description.** Test whether a string is lexically less than another string, based on the ASCII  
9       collating sequence.

10      **Class.** Elemental function.

11      **Arguments.**

12      `STRING_A`            must be of type default character.

13      `STRING_B`            must be of type default character.

14      **Result Type and Type Parameters.** Default logical.

15      **Result Value.** If the strings are of unequal length, the comparison is made as if the shorter  
16      string were extended on the right with blanks to the length of the longer string. If either  
17      string contains a character not in the ASCII character set, the result is processor dependent.  
18      The result is true if `STRING_A` precedes `STRING_B` in the ASCII collating sequence; other-  
19      wise, the result is false. The result is false if both `STRING_A` and `STRING_B` are of zero  
20      length.

21      **Example.** `LLT ('ONE', 'TWO')` has the value true.

### 22      **13.13.59 LOG (X).**

23      **Description.** Natural logarithm.

24      **Class.** Elemental function.

25      **Argument.** `X` must be of type real or complex. If `X` is real, its value must be greater than  
26      zero. If `X` is complex, its value must not be zero.

27      **Result Type and Type Parameter.** Same as `X`.

28      **Result Value.** The result has a value equal to a processor-dependent approximation to  $\log_e X$ .  
29      A result of type complex is the principal value with imaginary part  $\omega$  in the range  $-\pi < \omega \leq \pi$ .  
30      The imaginary part of the result is  $\pi$  only when the real part of the argument is less than zero  
31      and the imaginary part of the argument is zero.

32      **Example.** `LOG (10.0)` has the value 2.3025851 (approximately).

### 33      **13.13.60 LOG10 (X).**

34      **Description.** Common logarithm.

35      **Class.** Elemental function.

36      **Argument.** `X` must be of type real. The value of `X` must be greater than zero.

37      **Result Type and Type Parameter.** Same as `X`.

38      **Result Value.** The result has a value equal to a processor-dependent approximation to  
39       $\log_{10} X$ .

40      **Example.** `LOG10 (10.0)` has the value 1.0 (approximately).

1 **13.13.61 LOGICAL (L, KIND).**2 **Optional Argument.** KIND3 **Description.** Converts between kinds of logical.4 **Class.** Elemental function.5 **Arguments.**

6 L must be of type logical.

7 KIND (optional) must be a scalar integer initialization expression.

8 **Result Type and Type Parameter.** Logical. If KIND is present, the kind type parameter is  
9 that specified by KIND; otherwise, the kind type parameter is that of default logical.10 **Result Value.** The value is that of L.11 **Example.** LOGICAL (L .OR. NOT. L) has the value true and is of type default logical,  
12 regardless of the kind type parameter of the logical variable L.13 **13.13.62 MATMUL (MATRIX\_A, MATRIX\_B).**14 **Description.** Performs matrix multiplication of numeric or logical matrices.15 **Class.** Transformational function.16 **Arguments.**17 MATRIX\_A must be of numeric type (integer, real, or complex) or of logical type. It  
18 must be array valued and of rank one or two.19 MATRIX\_B must be of numeric type if MATRIX\_A is of numeric type and of logical  
20 type if MATRIX\_A is of logical type. It must be array valued and of  
21 rank one or two. If MATRIX\_A has rank one, MATRIX\_B must have  
22 rank two. If MATRIX\_B has rank one, MATRIX\_A must have rank two.  
23 The size of the first (or only) dimension of MATRIX\_B must equal the  
24 size of the last (or only) dimension of MATRIX\_A.25 **Result Type, Type Parameter, and Shape.** If the arguments are of numeric type, the type  
26 and type parameter of the result are determined by the types of the arguments according to  
27 7.1.4. If the arguments are of type logical, the result is of type logical with the type param-  
28 eters of the ~~expression MATRIX\_A .AND. MATRIX\_B~~ according to 7.1.4. The shape of the  
29 result depends on the shapes of the arguments as follows:30 *arguments*  
31 **Case (i):** If MATRIX\_A has shape  $(n, m)$  and MATRIX\_B has shape  $(m, k)$ , the result has  
32 shape  $(n, k)$ .32 **Case (ii):** If MATRIX\_A has shape  $(m)$  and MATRIX\_B has shape  $(m, k)$ , the result has  
33 shape  $(k)$ .34 **Case (iii):** If MATRIX\_A has shape  $(n, m)$  and MATRIX\_B has shape  $(m)$ , the result has  
35 shape  $(n)$ .36 **Result Value.**37 **Case (i):** Element  $(i, j)$  of the result has the value SUM (MATRIX\_A  $(i, :)$  \* MATRIX\_B  $(:, j)$ ) if the arguments are of numeric type and has the value ANY (MATRIX\_A  $(i, :)$  .AND. MATRIX\_B  $(:, j)$ ) if the arguments are of logical type.40 **Case (ii):** Element  $(j)$  of the result has the value SUM (MATRIX\_A  $(:, :)$  \* MATRIX\_B  $(:, j)$ ) if  
41 the arguments are of numeric type and has the value ANY (MATRIX\_A  $(:, :)$  .AND. MATRIX\_B  $(:, j)$ ) if the arguments are of logical type.42 **Case (iii):** Element  $(i)$  of the result has the value SUM (MATRIX\_A  $(i, :)$  \* MATRIX\_B  $(:)$ ) if  
43 the arguments are of numeric type and has the value ANY (MATRIX\_A  $(i, :)$  .AND. MATRIX\_B  $(:)$ ) if the arguments are of logical type.  
44  
45

1       **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]  
 2       and [1, 2, 3].

3       **Case (i):**       The result of MATMUL (A, B) is the matrix-matrix product AB with the value  
 4        $\begin{bmatrix} 14 & 20 \\ 20 & 29 \end{bmatrix}$ .

5       **Case (ii):**       The result of MATMUL (X, A) is the vector-matrix product XA with the value  
 6       [5, 8, 11].

7       **Case (iii):**       The result of MATMUL (A, Y) is the matrix-vector product AY with the value  
 8       [14, 20].

9       **13.13.63 MAX (A1, A2, A3, ...).**

10       **Optional Arguments.** A3, ...

11       **Description.** Maximum value.

12       **Class.** Elemental function.

13       **Arguments.** The arguments must all have the same type which must be integer or real and  
 14       they must all have the same type parameter.

15       **Result Type and Type Parameter.** Same as the arguments.

16       **Result Value.** The value of the result is that of the largest argument.

17       **Example.** MAX (-9.0, 7.0, 2.0) has the value 7.0.

18       **13.13.64 MAXEXPONENT (X).**

19       **Description.** Returns the maximum exponent in the model representing numbers of the  
 20       same type and type parameter as the argument.

21       **Class.** Inquiry function.

22       **Argument.** X must be of type real. It may be scalar or array valued.

23       **Result Type, Type Parameter, and Shape.** Default integer scalar.

24       **Result Value.** The result has the value  $e_{\max}$ , as defined in 13.7.1 for the model representing  
 25       numbers of the same type and type parameter as X.

26       **Example.** MAXEXPONENT (X) has the value 127 for real X whose model is as at the end of  
 27       13.7.1.

28       **13.13.65 MAXLOC (ARRAY, MASK).**

29       **Optional Argument.** MASK

30       **Description.** Determine the location of the first element of ARRAY having the maximum  
 31       value of the elements identified by MASK.

32       **Class.** Transformational function.

33       **Arguments.**

34       **ARRAY**               must be of type integer or real. It must not be scalar.

35       **MASK (optional)**    must be of type logical and must be conformable with ARRAY.

36       **Result Type, Type Parameter, and Shape.** The result is of type default integer; it is an array  
 37       of rank one and of size equal to the rank of ARRAY.

1 **Result Value.**

2 *Case (i):* If MASK is absent, the result is a rank-one array whose element values are the  
 3 values of the subscripts of an element of ARRAY whose value equals the maxi-  
 4 mum value of all of the elements of ARRAY. The  $i$ th subscript returned lies in  
 5 the range 1 to  $e_i$ , where  $e_i$  is the extent of the  $i$ th dimension of ARRAY. If more  
 6 than one element has the maximum value, the element whose subscripts are  
 7 returned is the first such element, taken in array element order. If ARRAY has  
 8 size zero, the value of the result is processor dependent.

9 *Case (ii):* If MASK is present, the result is a rank-one array whose element values are the  
 10 values of the subscripts of an element of ARRAY, corresponding to a true ele-  
 11 ment of MASK, whose value equals the maximum value of all such elements of  
 12 ARRAY. The  $i$ th subscript returned lies in the range 1 to  $e_i$ , where  $e_i$  is the extent  
 13 of the  $i$ th dimension of ARRAY. If more than one such element has the maxi-  
 14 mum value, the element whose subscripts are returned is the first such element  
 15 taken in array element order. If there are no such elements (that is, if ARRAY  
 16 has size zero or every element of MASK has the value false), the value of the  
 17 result is processor dependent.

18 **Examples.**

19 *Case (i):* The value of MAXLOC (/ 2, 4, 6 /) is [3].

20 *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  
 21 [3, 2]. Note that this is true even if A has a declared lower bound other than 1.

22 **13.13.66 MAXVAL (ARRAY, DIM, MASK).**

23 **Optional Arguments.** DIM, MASK

24 **Description.** Maximum value of the elements of ARRAY along dimension DIM correspond-  
 25 ing to the true elements of MASK.

26 **Class.** Transformational function.

27 **Arguments.**

28 ARRAY must be of type integer or real. It must not be scalar.

29 DIM (optional) must be scalar and of type integer with a value in the range  $1 \leq \text{DIM} \leq n$ ,  
 30 where  $n$  is the rank of ARRAY.

31 MASK (optional) must be of type logical and must be conformable with ARRAY.

32 **Result Type, Type Parameter, and Shape.** The result is of the same type and type parameter  
 33 as ARRAY. It is scalar if DIM is absent or ARRAY has rank one; otherwise, the result is an  
 34 array of rank  $n-1$  and of shape  $(d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n)$  where  $(d_1, d_2, \dots, d_n)$  is the  
 35 shape of ARRAY.

36 **Result Value.**

37 *Case (i):* The result of MAXVAL (ARRAY) has a value equal to the maximum value of all  
 38 the elements of ARRAY or has the value of the negative number of the largest  
 39 magnitude supported by the processor for numbers of the type and type param-  
 40 eter of ARRAY if ARRAY has size zero.

41 *Case (ii):* The result of MAXVAL (ARRAY, MASK = MASK) has a value equal to the max-  
 42 imum value of the elements of ARRAY corresponding to true elements of  
 43 MASK or has the value of the negative number of the largest magnitude sup-  
 44 ported by the processor for numbers of the type and type parameter of ARRAY  
 45 if there are no true elements.

## Intrinsic Procedures

1        *Case (iii):* If ARRAY has rank one, MAXVAL (ARRAY, DIM [,MASK]) has a value equal to  
2        that of MAXVAL (ARRAY [,MASK = MASK]). Otherwise, the value of element  
3        ( $s_1, s_2, \dots, s_{DIM-1}, s_{DIM+1}, \dots, s_n$ ) of MAXVAL (ARRAY, DIM [,MASK]) is equal to  
4        MAXVAL (ARRAY ( $s_1, s_2, \dots, s_{DIM-1}, :, s_{DIM+1}, \dots, s_n$ ), [, MASK = MASK ( $s_1, s_2, \dots,$   
5         $s_{DIM-1}, :, s_{DIM+1}, \dots, s_n$ ) ]).

6        **Examples.**

7        *Case (i):* The value of MAXVAL ((/ 1, 2, 3 /)) is 3.

8        *Case (ii):* MAXVAL (C, MASK = C .LT. 0.0) finds the maximum of the negative elements  
9        of C.

10       *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,  
11       DIM = 2) is [5, 6].

### 12    13.13.67 MERGE (TSOURCE, FSOURCE, MASK).

13       **Description.** Choose alternative value according to the value of a mask.

14       **Class.** Elemental function.

15       **Arguments.**

16       TSOURCE            may be of any type.

17       FSOURCE            must be of the same type and type parameters as TSOURCE.

18       MASK                must be of type logical.

19       **Result Type and Type Parameters.** Same as TSOURCE.

20       **Result Value.** The result is TSOURCE if MASK is true and FSOURCE otherwise.

21       **Example.** If TSOURCE is the array  $\begin{bmatrix} 1 & 6 & 5 \\ 2 & 6 \end{bmatrix}$ , FSOURCE is the array  $\begin{bmatrix} 0 & 3 & 2 \\ 7 & 4 & 8 \end{bmatrix}$  and MASK is

22       the array  $\begin{bmatrix} T & . & T \\ . & . & T \end{bmatrix}$ , where "T" represents true and "." represents false, then MERGE

23       (TSOURCE, FSOURCE, MASK) is  $\begin{bmatrix} 1 & 3 & 5 \\ 7 & 4 & 6 \end{bmatrix}$ . The value of MERGE (1.0, 0.0, K > 0) is 1.0 for K  
24       = 5 and 0.0 for K = -2.

### 25    13.13.68 MIN (A1, A2, A3, ...).

26       **Optional Arguments.** A3, ...

27       **Description.** Minimum value.

28       **Class.** Elemental function.

29       **Arguments.** The arguments must all be of the same type which must be integer or real and  
30       they must all have the same type parameter.

31       **Result Type and Type Parameter.** Same as the arguments.

32       **Result Value.** The value of the result is that of the smallest argument.

33       **Example.** MIN (-9.0, 7.0, 2.0) has the value -9.0.

### 34    13.13.69 MINEXPONENT (X).

35       **Description.** Returns the minimum (most negative) exponent in the model representing  
36       numbers of the same type and type parameter as the argument.



- 1 **Class.** Inquiry function.  
 2 **Argument.** X must be of type real. It may be scalar or array valued.  
 3 **Result Type, Type Parameter, and Shape.** Default integer scalar.  
 4 **Result Value.** The result has the value  $e_{\min}$ , as defined in 13.7.1 for the model representing  
 5 numbers of the same type and type parameter as X.  
 6 **Example.** MINEXPONENT (X) has the value -126 for real X whose model is as at the end of  
 7 13.7.1.

### 8 13.13.70 MINLOC (ARRAY, MASK).

9 **Optional Argument.** MASK

10 **Description.** Determine the location of the first element of ARRAY having the minimum  
 11 value of the elements identified by MASK.

12 **Class.** Transformational function.

13 **Arguments.**

14 ARRAY must be of type integer or real. It must not be scalar.

15 MASK (optional) must be of type logical and must be conformable with ARRAY.

16 **Result Type, Type Parameter, and Shape.** The result is of type default integer; it is an array  
 17 of rank one and of size equal to the rank of ARRAY.

18 **Result Value.**

19 *Case (i):* If MASK is absent, the result is a rank-one array whose element values are the  
 20 values of the subscripts of an element of ARRAY whose value equals the mini-  
 21 mum value of all the elements of ARRAY. The  $i$ th subscript returned lies in the  
 22 range 1 to  $e_i$ , where  $e_i$  is the extent of the  $i$ th dimension of ARRAY. If more than  
 23 one element has the minimum value, the element whose subscripts are returned  
 24 is the first such element, taken in array element order. If ARRAY has size zero,  
 25 the value of the result is processor dependent.

26 *Case (ii):* If MASK is present, the result is a rank-one array whose element values are the  
 27 values of the subscripts of an element of ARRAY, corresponding to a true ele-  
 28 ment of MASK, whose value equals the minimum value of all such elements of  
 29 ARRAY. The  $i$ th subscript returned lies in the range 1 to  $e_i$ , where  $e_i$  is the extent  
 30 of the  $i$ th dimension of ARRAY. If more than one such element has the mini-  
 31 mum value, the element whose subscripts are returned is the first such element  
 32 taken in array element order. If ARRAY has size zero or every element of  
 33 MASK has the value false, the value of the result is processor dependent.

34 **Examples.**

35 *Case (i):* The value of MINLOC ((/ 2, 4, 6 /)) is [1].

36 *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  
 37 [1, 4]. Note that this is true even if A has a declared lower bound other than 1.

### 38 13.13.71 MINVAL (ARRAY, DIM, MASK).

39 **Optional Arguments.** DIM, MASK

40 **Description.** Minimum value of all the elements of ARRAY along dimension DIM corre-  
 41 sponding to true elements of MASK.

## Intrinsic Procedures

- 1        **Class.** Transformational function.
- 2        **Arguments.**
- 3        **ARRAY**                must be of type integer or real. It must not be scalar.
- 4        **DIM (optional)**        must be scalar and of type integer with a value in the range  $1 \leq \text{DIM} \leq n$ ,  
5                                    where  $n$  is the rank of **ARRAY**.
- 6        **MASK (optional)**        must be of type logical and must be conformable with **ARRAY**.
- 7        **Result Type, Type Parameter, and Shape.** The result is of the same type and type parameter  
8        as **ARRAY**. It is scalar if **DIM** is absent or **ARRAY** has rank one; otherwise, the result is an  
9        array of rank  $n-1$  and of shape  $(d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n)$  where  $(d_1, d_2, \dots, d_n)$  is the  
10       shape of **ARRAY**.
- 11       **Result Value.**
- 12       *Case (i):*        The result of **MINVAL (ARRAY)** has a value equal to the minimum value of all  
13       the elements of **ARRAY** or has the value of the positive number of the largest  
14       magnitude supported by the processor for numbers of the type and type param-  
15       eters of **ARRAY** if **ARRAY** has size zero.
- 16       *Case (ii):*        The result of **MINVAL (ARRAY, MASK = MASK)** has a value equal to the mini-  
17       mum value of the elements of **ARRAY** corresponding to true elements of **MASK**  
18       or has the value of the positive number of the largest magnitude supported by  
19       the processor for numbers of the type and type parameters of **ARRAY** if there  
20       are no true elements.
- 21       *Case (iii):*        If **ARRAY** has rank one, **MINVAL (ARRAY, DIM [,MASK])** has a value equal to  
22       that of **MINVAL (ARRAY [,MASK = MASK])**. Otherwise, the value of element  
23        $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$  of **MINVAL (ARRAY, DIM [,MASK])** is equal to  
24       **MINVAL (ARRAY  $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$  [, MASK= MASK  $(s_1, s_2, \dots,$   
25        $s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$  ] )**.
- 26       **Examples.**
- 27       *Case (i):*        The value of **MINVAL ((/ 1, 2, 3 /))** is 1.
- 28       *Case (ii):*        **MINVAL (C, MASK = C .GT. 0.0)** forms the minimum of the positive elements  
29       of **C**.
- 30       *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,**  
31       **DIM = 2)** is [1, 2].

### 32 13.13.72 MOD (A, P).

- 33        **Description.** Remainder function.
- 34        **Class.** Elemental function.
- 35        **Arguments.**
- 36        **A**                        must be of type integer or real.
- 37        **P**                        must be of the same type and kind type parameter as **A**.
- 38        **Result Type and Type Parameter.** Same as **A**.
- 39        **Result Value.** If  $P \neq 0$ , the value of the result is  $A - \text{INT}(A/P) * P$ . If  $P = 0$ , the result is proc-  
40        essor defined. *dependent.*
- 41        **Examples.** **MOD (3.0, 2.0)** has the value 1.0. *approximately* **MOD (8, 5)** has the value 3. **MOD (-8, 5)** has the  
42        value -3. **MOD (8, -5)** has the value 3. **MOD (-8, -5)** has the value -3.

## 1 13.13.73 MODULO (A, P).

2 Description. Modulo function.

3 Class. Elemental function.

4 Arguments.

5 A must be of type integer or real.

6 P must be of the same type and kind type parameter as A.

7 Result Type and Type Parameter. Same as A.

8 Result Value.

9 Case (i): A is of type integer. If  $P \neq 0$ , the value of the result is  $A - \text{FLOOR}(\text{REAL}(A) /$   
 10  $\text{REAL}(P)) * P$ . If  $P = 0$ , the result is processor defined. ~~dependent.~~

11 Case (ii): A is of type real. If  $P \neq 0$ , the value of the result is  $A - \text{FLOOR}(A / P) * P$ . If  $P =$   
 12  $0$ , the result is processor defined. ~~dependent.~~

13 Examples. MODULO (8, 5) has the value 3. MODULO (-8, 5) has the value 2. MODULO (8,  
 14 -5) has the value -2. MODULO (-8, -5) has the value -3.

## 15 13.13.74 MVBITS (FROM, FROMPOS, LEN, TO, TOPOS).

16 Description. Copies a sequence of bits from one data object to another.

17 Class. Elemental subroutine.

18 Arguments.

19 FROM must be of type integer. It is an INTENT (IN) argument.

20 FROMPOS must be of type integer and nonnegative. It is an INTENT (IN) argu-  
 21 ment. FROMPOS + LEN must be less than or equal to BIT\_SIZE  
 22 (FROM). The model for the interpretation of an integer value as a  
 23 sequence of bits is in 13.5.7.

24 LEN must be of type integer and nonnegative. It is an INTENT (IN) argu-  
 25 ment.

26 TO must be a variable of type integer with the same kind type parameter  
 27 value as FROM and may be the same variable as FROM. It is an  
 28 INTENT (INOUT) argument. TO is set by copying the sequence of bits  
 29 of length LEN, starting at position FROMPOS of FROM to positions  
 30 TOPOS of TO. No other bits of TO are altered. On return, the LEN bits  
 31 of TO starting at TOPOS are equal to the value that the LEN bits of  
 32 FROM starting at FROMPOS had on entry. The model for the interpre-  
 33 tation of an integer value as a sequence of bits is in 13.5.7.

34 TOPOS must be of type integer and nonnegative. It is an INTENT (IN) argu-  
 35 ment. TOPOS + LEN must be less than or equal to BIT\_SIZE (TO).

36 Example. If TO has the initial value 6, the value of TO after the statement CALL MVBITS (7,  
 37 2, 2, TO, 0) is 5.

## 38 13.13.75 NEAREST (X, S).

39 Description. Returns the nearest different machine representable number in a given direc-  
 40 tion.

41 Class. Elemental function.

42 Arguments.

43 X must be of type real.

## Intrinsic Procedures

- 1        **S**                    must be of type real and not equal to zero.
- 2        **Result Type and Type Parameter.** Same as X.
- 3        **Result Value.** The result has a value equal to the machine representable number distinct
- 4        from X and nearest to it in the direction of the infinity with the same sign as S.
- 5        **Example.** NEAREST (3.0, 2.0) has the value  $3+2^{-22}$  on a machine whose representation is that
- 6        of the model at the end of 13.7.1.

### 7        **13.13.76 NINT (A, KIND).**

8        **Optional Argument.** KIND

9        **Description.** Nearest integer.

10       **Class.** Elemental function.

11       **Arguments.**

12       A                    must be of type real.

13       KIND (optional)    must be a scalar integer initialization expression.

14       **Result Type and Type Parameter.** Integer. If KIND is present, the kind type parameter is

15       that specified by KIND; otherwise, the kind type parameter is that of default integer type.

16       **Result Value.** If  $A > 0$ , NINT (A) has the value INT (A+0.5); if  $A \leq 0$ , NINT (A) has the value

17       INT (A-0.5). *The result is undefined if the processor cannot represent the result in the specified integer type.*

18       **Example.** NINT (2.783) has the value 3.

### 19       **13.13.77 NOT (I).**

20       **Description.** Performs a logical complement.

21       **Class.** Elemental function.

22       **Argument.** I must be of type integer.

23       **Result Type and Type Parameter.** Same as I.

24       **Result Value.** The result has the value obtained by complementing I bit-by-bit according to

25       the following truth table:

| I | NOT (I) |
|---|---------|
| 1 | 0       |
| 0 | 1       |

30       The model for the interpretation of an integer value as a sequence of bits is in 13.5.7.

31       **Example.** If I is represented by the string of bits 01010101, NOT (I) has the binary value

32       10101010.

### 33       **13.13.78 PACK (ARRAY, MASK, VECTOR).**

34       **Optional Argument.** VECTOR

35       **Description.** Pack an array into an array of rank one under the control of a mask.

36       **Class.** Transformational function.

37       **Arguments.**

38       ARRAY                may be of any type. It must not be scalar.

39       MASK                 must be of type logical and must be conformable with ARRAY.

1 VECTOR (optional) must be of the same type and type parameters as ARRAY and must  
 2 have rank one. VECTOR must have at least as many elements as there  
 3 are true elements in MASK. If MASK is scalar with the value true, VEC-  
 4 TOR must have at least as many elements as there are in ARRAY.

5 **Result Type, Type Parameter, and Shape.** The result is an array of rank one with the same  
 6 type and type parameters as ARRAY. If VECTOR is present, the result size is that of VEC-  
 7 TOR; otherwise, the result size is the number  $t$  of true elements in MASK unless MASK is sca-  
 8 lar with the value true, in which case the result size is the size of ARRAY.

9 **Result Value.** Element  $i$  of the result is the element of ARRAY that corresponds to the  $i$ th  
 10 true element of MASK, taking elements in array element order, for  $i = 1, 2, \dots, t$ . If VECTOR is  
 11 present and has size  $n > t$ , element  $i$  of the result has the value VECTOR ( $i$ ), for  $i = t + 1, \dots, n$ .

12 **Example.** The nonzero elements of an array M with the value  $\begin{bmatrix} 0 & 0 & 0 \\ 9 & 0 & 0 \\ 0 & 0 & 7 \end{bmatrix}$  may be "gathered" by  
 13 the function PACK. The result of PACK (M, MASK = M .NE. 0) is [9, 7] and the result of  
 14 PACK (M, M .NE. 0, VECTOR = (/ 2, 4, 6, 8, 10, 12 /)) is [9, 7, 6, 8, 10, 12].

15 **13.13.79 PRECISION (X).**

16 **Description.** Returns the decimal precision in the model representing real numbers with the  
 17 same kind type parameter as the argument.

18 **Class.** Inquiry function.

19 **Argument.** X must be of type real or complex. It may be scalar or array valued.

20 **Result Type, Type Parameter, and Shape.** Default integer scalar.

21 **Result Value.** The result has the value  $\text{INT} ((p-1) * \text{LOG}_{10}(b)) + k$ , where  $b$  and  $p$  are as  
 22 defined in 13.7.1 for the model representing real numbers with the same value for the kind  
 23 type parameter as X, and where  $k$  is 1 if  $b$  is an integral power of 10 and 0 otherwise.

24 **Example.** PRECISION (X) has the value  $\text{INT} (23 * \text{LOG}_{10}(2.)) = \text{INT} (6.92\dots) = 6$  for real X  
 25 whose model is as at the end of 13.7.1.

26 **13.13.80 PRESENT (A).**

27 **Description.** Determine whether an optional argument is present.

28 **Class.** Inquiry function.

29 **Argument.** A must be an optional argument of the procedure in which the PRESENT func-  
 30 tion reference appears.

31 **Result Type and Type Parameters.** Default logical scalar.

32 **Result Value.** The result has the value true if A is present (12.5.2.8) and otherwise has the  
 33 value false.

34 **13.13.81 PRODUCT (ARRAY, DIM, MASK).**

35 **Optional Arguments.** DIM, MASK

36 **Description.** Product of all the elements of ARRAY along dimension DIM corresponding to  
 37 the true elements of MASK.

38 **Class.** Transformational function.

39 **Arguments.**

40 **ARRAY** must be of type integer, real, or complex. It must not be scalar.

## Intrinsic Procedures

1 DIM (optional) must be scalar and of type integer with a value in the range  $1 \leq \text{DIM} \leq n$ ,  
2 where  $n$  is the rank of ARRAY.

3 MASK (optional) must be of type logical and must be conformable with ARRAY.

4 **Result Type, Type Parameter, 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  
5 an array of rank  $n-1$  and of shape  $(d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n)$  where  $(d_1, d_2, \dots, d_n)$  is the  
6 shape of ARRAY.  
7

8 **Result Value.**

9 *Case (i):* The result of PRODUCT (ARRAY) has a value equal to a processor-dependent  
10 approximation to the product of all the elements of ARRAY or has the value one  
11 if ARRAY has size zero.

12 *Case (ii):* The result of PRODUCT (ARRAY, MASK = MASK) has a value equal to a  
13 processor-dependent approximation to the product of the elements of ARRAY  
14 corresponding to the true elements of MASK or has the value one if there are no  
15 true elements.

16 *Case (iii):* If ARRAY has rank one, PRODUCT (ARRAY, DIM [,MASK]) has a value equal  
17 to that of PRODUCT (ARRAY [,MASK = MASK]). Otherwise, the value of element  
18  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$  of PRODUCT (ARRAY, DIM [,MASK]) is  
19 equal to PRODUCT (ARRAY  $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$  [, MASK = MASK  
20  $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$  ]).

21 **Examples.**

22 *Case (i):* The value of PRODUCT ((/ 1, 2, 3 /)) is 6.

23 *Case (ii):* PRODUCT (C, MASK = C.GT. 0.0) forms the product of the positive elements of  
24 C.

25 *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  
26 (B, DIM = 2) is [15, 48].

### 27 13.13.82 RADIX (X).

28 **Description.** Returns the base of the model representing numbers of the same type and type  
29 parameter as the argument.

30 **Class.** Inquiry function.

31 **Argument.** X must be of type integer or real. It may be scalar or array valued.

32 **Result Type, Type Parameter, and Shape.** Default integer scalar.

33 **Result Value.** The result has the value  $r$  if X is of type integer and the value  $b$  if X is of type  
34 real, where  $r$  and  $b$  are as defined in 13.7.1 for the model representing numbers of the same  
35 type and type parameter as X.

36 **Example.** RADIX (X) has the value 2 for real X whose model is as at the end of 13.7.1.

### 37 13.13.83 RANDOM\_NUMBER (HARVEST).

38 **Description.** Returns one pseudorandom number or an array of pseudorandom numbers  
39 from the uniform distribution over the range  $0 \leq x < 1$ .

40 **Class.** Subroutine.

41 **Argument.** HARVEST must be of type real. It is an INTENT (OUT) argument. It may be a  
42 scalar or an array variable. It is set to contain pseudorandom numbers from the uniform dis-  
43 tribution in the interval  $0 \leq x < 1$ .

```

1  Examples.
2      REAL X, Y (10, 10)
3      ! Initialize X with a pseudorandom number
4      CALL RANDOM_NUMBER (HARVEST = X)
5      CALL RANDOM_NUMBER (Y)
6      ! X and Y contain uniformly distributed random numbers

```

### 7 13.13.84 RANDOM\_SEED (SIZE, PUT, GET).

8 **Optional Arguments.** SIZE, PUT, GET

9 **Description.** Restarts or queries the pseudorandom number generator used by RANDOM\_NUMBER.

10 **Class.** Subroutine.

11 **Arguments.** There must either be exactly one or no arguments present.

12 **SIZE (optional)** must be scalar and of type default integer. It is an INTENT (OUT) argu-  
13 ment. It is set to the number  $N$  of integers that the processor uses to  
14 hold the value of the seed.

15 **PUT (optional)** must be a default integer array of rank one and size  $\geq N$ . It is an  
16 INTENT (IN) argument. It is used by the processor to set the seed  
17 value.

18 **GET (optional)** must be a default integer array of rank one and size  $\geq N$ . It is an  
19 INTENT (OUT) argument. It is set by the processor to the current value  
20 of the seed.

21 If no argument is present, the processor sets the seed to a processor-dependent value.

22 **Examples.**

```

23      CALL RANDOM_SEED                                ! Processor initialization
24      CALL RANDOM_SEED (SIZE = K)                    ! Sets K = N
25      CALL RANDOM_SEED (PUT = SEED (1 : K))          ! Set user seed
26      CALL RANDOM_SEED (GET = OLD (1 : K))           ! Read current seed

```

### 27 13.13.85 RANGE (X).

28 **Description.** Returns the decimal exponent range in the model representing integer or real  
29 numbers with the same kind type parameter as the argument.

30 **Class.** Inquiry function.

31 **Argument.** X must be of type integer, real, or complex. It may be scalar or array valued.

32 **Result Type, Type Parameter, and Shape.** Default integer scalar.

33 **Result Value.**

34 **Case (i):** For an integer argument, the result has the value INT (LOG10 (*huge*)), where  
35 *huge* is the largest positive integer in the model representing integer numbers  
36 with same kind type parameter as X (13.7.1).

37 **Case (ii):** For a real or complex argument, the result has the value INT (MIN (LOG10  
38 (*huge*), -LOG10 (*tiny*))), where *huge* and *tiny* are the largest and smallest positive  
39 numbers in the model representing real numbers with the same value for the  
40 kind type parameter as X (13.7.1).

41 **Example.** RANGE (X) has the value 38 for real X whose model is as at the end of 13.7.1, since  
42 in this case  $huge = (1-2^{-24}) \times 2^{127}$  and  $tiny = 2^{-127}$ .

## Intrinsic Procedures

### 1 13.13.86 REAL (A, KIND).

2 Optional Argument. KIND

3 Description. Convert to real type.

4 Class. Elemental function.

5 Arguments.

6 A must be of type integer, real, or complex.

7 KIND (optional) must be a scalar integer initialization expression.

8 Result Type and Type Parameter. Real.

9 Case (i): If A is of type integer or real and KIND is present, the type parameter is that  
10 specified by KIND. If A is of type integer or real and KIND is not present, the  
11 type parameter is the processor-dependent type parameter for the default real  
12 type.

13 Case (ii): If A is of type complex and KIND is present, the type parameter is that specified  
14 by KIND. If A is of type complex and KIND is not present, the type parameter  
15 is the type parameter of A.

16 Result Value.

17 Case (i): If A is of type integer or real, the result is equal to a processor-dependent  
18 approximation to A.

19 Case (ii): If A is of type complex, the result is equal to a processor-dependent approxima-  
20 tion to the real part of A.

21 Examples. REAL (-3) has the value -3.0. REAL (Z) has the same kind type parameter and  
22 the same value as the real part of the complex variable Z.

### 23 13.13.87 REPEAT (STRING, NCOPIES).

24 Description. Concatenate several copies of a string.

25 Class. Transformational function.

26 Arguments.

27 STRING must be scalar and of type character.

28 NCOPIES must be scalar and of type integer. Its value must not be negative.

29 Result Type, Type Parameter, and Shape. Character scalar of length NCOPIES times that of  
30 STRING, with the same kind type parameter as STRING.

31 Result Value. The value of the result is the concatenation of NCOPIES copies of STRING.

32 Examples. REPEAT ('H', 2) has the value HH. REPEAT ('XYZ', 0) has the value of a zero-  
33 length string.

### 34 13.13.88 RESHAPE (SOURCE, SHAPE, PAD, ORDER).

35 Optional Arguments. PAD, ORDER

36 Description. ~~Change the shape of an array.~~ Constructs an array of a specified shape  
37 from the elements of a given array.

38 Class. Transformational function.

39 Arguments.

40 SOURCE may be of any type. It must be array valued. If PAD is absent or of size  
41 zero, the size of SOURCE must be  $\geq$  PRODUCT (SHAPE). The size of  
the result is the product of the values of the elements of SHAPE.



- 1        **SHAPE**                must be of type integer, rank one, and constant size. Its size must be
- 2                                positive and less than 8. It must not have an element whose value is
- 3                                negative.
- 4        **PAD (optional)**        must be of the same type and type parameters as **SOURCE**. **PAD** must
- 5                                be array valued.
- 6        **ORDER (optional)**        must be of type integer, must have the same shape as **SHAPE**, and its
- 7                                value must be a permutation of (1, 2, ..., *n*), where *n* is the size of
- 8                                **SHAPE**. If absent, it is as if it were present with value (1, 2, ..., *n*).

9        **Result Type, Type Parameter, and Shape.** The result is an array of shape **SHAPE** (that is,  
 10        **SHAPE (RESHAPE (SOURCE, SHAPE, PAD, ORDER))** is equal to **SHAPE**) with the same  
 11        type and type parameters as **SOURCE**.

12        **Result Value.** The elements of the result, taken in permuted subscript order **ORDER** (1), ...,  
 13        **ORDER** (*n*), are those of **SOURCE** in normal array element order followed if necessary by  
 14        those of **PAD** in array element order, followed if necessary by additional copies of **PAD** in  
 15        array element order.

16        **Examples.** **RESHAPE** ((/ 1, 2, 3, 4, 5, 6 /), (/ 2, 3 /)) has the value  $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ . **RESHAPE**  
 17        ((/ 1, 2, 3, 4, 5, 6 /), (/ 2, 4 /), (/ 0, 0 /), (/ 2, 1 /)) has the value  $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 0 & 0 \end{bmatrix}$ .

18        **13.13.89 RRSPACING (X).**

19        **Description.** Returns the reciprocal of the relative spacing of model numbers near the argu-  
 20        ment value.

21        **Class.** Elemental function.

22        **Argument.** **X** must be of type real.

23        **Result Type and Type Parameter.** Same as **X**.

24        **Result Value.** The result has the value  $|X \times b^{-e}| \times b^p$ , where *b*, *e*, and *p* are as defined in  
 25        13.7.1 for the model representation of **X**.

26        **Example.** **RRSPACING** (-3.0) has the value  $0.75 \times 2^{24}$  for reals whose model is as at the end of  
 27        13.7.1.

28        **13.13.90 SCALE (X, I).**

29        **Description.** Returns  $X \times b^I$  where *b* is the base in the model representation of **X**.

30        **Class.** Elemental function.

31        **Arguments.**

32        **X**                        must be of type real.

33        **I**                         must be of type integer.

34        **Result Type and Type Parameter.** Same as **X**.

35        **Result Value.** The result has the value  $X \times b^I$ , where *b* is defined in 13.7.1 for model numbers  
 36        representing values of **X**, provided this result is within range; *if not, the result is processor*

37        **Example.** **SCALE** (3.0, 2) has the value 12.0 for reals whose model is as at the end of 13.7.1.

38        **13.13.91 SCAN (STRING, SET, BACK).**

39        **Optional Argument.** **BACK**

## Intrinsic Procedures

1       **Description.** Scan a string for any one of the characters in a set of characters.

2       **Class.** Elemental function.

3       **Arguments.**

4       STRING            must be of type character.

5       SET                must be of type character with the same kind type parameter as  
6       STRING.

7       BACK (optional)   must be of type logical.

8       **Result Type and Type Parameter.** Default integer.

9       **Result Value.**

10      *Case (i):*        If BACK is absent or is present with the value false and if STRING contains at  
11                        least one character that is in SET, the value of the result is the position of the left-  
12                        most character of STRING that is in SET.

13      *Case (ii):*       If BACK is present with the value true and if STRING contains at least one char-  
14                        acter that is in SET, the value of the result is the position of the rightmost charac-  
15                        ter of STRING that is in SET.

16      *Case (iii):*     The value of the result is zero if no character of STRING is in SET or if the length  
17                        of STRING or SET is zero.

18      **Examples.**

19      *Case (i):*        SCAN ('FORTRAN', 'TR') has the value 3.

20      *Case (ii):*       SCAN ('FORTRAN', 'TR', BACK = .TRUE.) has the value 5.

21      *Case (iii):*     SCAN ('FORTRAN', 'BCD') has the value 0.

### 22   13.13.92 SELECTED\_INT\_KIND (R).

23      **Description.** Returns a value of the kind type parameter of an integer data type that repre-  
24                        sents all integer values  $n$  with  $-10^R < n < 10^R$ .

25      **Class.** Transformational function.

26      **Arguments.**

27      R                    must be scalar and of type integer.

28      **Result Type, Type Parameter, and Shape.** Default integer scalar.

29      **Result Value.** The result has a value equal to the value of the kind type parameter of an inte-  
30                        ger data type that represents all values  $n$  in the range of values  $n$  with  $-10^R < n < 10^R$ , or if no  
31                        such kind type parameter is available on the processor, the result is  $-1$ . If more than one kind  
32                        type parameter meets the criteria, the value returned is the one with the smallest decimal  
33                        exponent range, unless there are several such values, in which case the smallest of these kind  
34                        values is returned.

35      **Example.** SELECTED\_INT\_KIND (6) has the value KIND (0) on a machine that supports  
36      default integer representation method with  $r = 2$  and  $q = 31$ .

### 37   13.13.93 SELECTED\_REAL\_KIND (P, R).

38      **Optional Arguments.** P, R

39      **Description.** Returns a value of the kind type parameter of a real data type with decimal  
40                        precision of at least P digits and a decimal exponent range of at least R.

41      **Class.** Transformational function.

42      **Arguments.** At least one argument must be present.

1 P (optional) must be scalar and of type integer.

2 R (optional) must be scalar and of type integer.

3 **Result Type, Type Parameter, and Shape.** Default integer scalar.

4 **Result Value.** The result has a value equal to a value of the kind type parameter of a real  
5 data type with decimal precision, as returned by the function PRECISION, of at least P digits  
6 and a decimal exponent range, as returned by the function RANGE, of at least R, or if no such  
7 kind type parameter is available on the processor, the result is -1 if the precision is not avail-  
8 able, -2 if the exponent range is not available, and -3 if neither is available. If more than one  
9 kind type parameter value meets the criteria, the value returned is the one with the smallest  
10 decimal precision, unless there are several such values, in which case the smallest of these  
11 kind values is returned.

12 **Example.** `SELECTED_REAL_KIND (6, 70)` has the value `KIND (0.0)` on a machine that sup-  
13 ports default real approximation method with  $b = 16$ ,  $p = 6$ ,  $e_{\min} = -64$ , and  $e_{\max} = 63$ .

#### 14 13.13.94 SET\_EXPONENT (X, I).

15 **Description.** Returns the model number whose fractional part is the fractional part of the  
16 model representation of X and whose exponent part is I.

17 **Class.** Elemental function.

18 **Arguments.**

19 X must be of type real.

20 I must be of type integer.

21 **Result Type and Type Parameter.** Same as X.

22 **Result Value.** The result has the value  $X \times b^{I-e}$ , where  $b$  and  $e$  are as defined in 13.7.1 for the  
23 model representation of X, provided this result is within range. If X has value zero, the result  
24 has value zero. *If not, the result is processor dependent.*

25 **Example.** `SET_EXPONENT (3.0, 1)` has the value 1.5 for reals whose model is as at the end of  
26 13.7.1.

#### 27 13.13.95 SHAPE (SOURCE).

28 **Description.** Returns the shape of an array or a scalar.

29 **Class.** Inquiry function.

30 **Argument.** SOURCE may be of any type. It may be array valued or scalar. It must not be a  
31 pointer that is disassociated or an allocatable array that is not allocated. It must not be an  
32 assumed-size array.

33 **Result Type, Type Parameter, and Shape.** The result is a default integer array of rank one  
34 whose size is equal to the rank of SOURCE.

35 **Result Value.** The value of the result is the shape of SOURCE.

36 **Examples.** The value of `SHAPE (A (2:5, -1:1))` is [4, 3]. The value of `SHAPE (3)` is the rank-  
37 one array of size zero.

#### 38 13.13.96 SIGN (A, B).

39 **Description.** Absolute value of A times the sign of B.

40 **Class.** Elemental function.

41 **Arguments.**

42 A must be of type integer or real.

## Intrinsic Procedures

1        **B**                    must be of the same type and kind type parameter as A.

2        **Result Type and Type Parameter.** Same as A.

3        **Result Value.** The value of the result is  $|A|$  if  $B \geq 0$  and  $-|A|$  if  $B < 0$ .

4        **Example.** SIGN (-3.0, 2.0) has the value 3.0.

### 5        **13.13.97 SIN (X).**

6        **Description.** Sine function.

7        **Class.** Elemental function.

8        **Argument.** X must be of type real or complex.

9        **Result Type and Type Parameter.** Same as X.

10       **Result Value.** The result has a value equal to a processor-dependent approximation to  $\sin(X)$ . If X is of type real, it is regarded as a value in radians. If X is of type complex, its real part is regarded as a value in radians.

13       **Example.** SIN (1.0) has the value 0.84147098 (approximately).

### 14       **13.13.98 SINH (X).**

15       **Description.** Hyperbolic sine function.

16       **Class.** Elemental function.

17       **Argument.** X must be of type real.

18       **Result Type and Type Parameter.** Same as X.

19       **Result Value.** The result has a value equal to a processor-dependent approximation to  $\sinh(X)$ .

21       **Example.** SINH (1.0) has the value 1.1752012 (approximately).

### 22       **13.13.99 SIZE (ARRAY, DIM).**

23       **Optional Argument.** DIM

24       **Description.** Returns the extent of an array along a specified dimension or the total number of elements in the array.

26       **Class.** Inquiry function.

27       **Arguments.**

28       **ARRAY**                may be of any type. It must not be scalar. It must not be a pointer that is disassociated or an allocatable array that is not allocated. If ARRAY is an assumed-size array, DIM must be present with a value less than the rank of ARRAY.

32       **DIM (optional)**        must be scalar and of type integer with a value in the range  $1 \leq \text{DIM} \leq n$ , where  $n$  is the rank of ARRAY.

34       **Result Type, Type Parameter, and Shape.** Default integer scalar.

35       **Result Value.** The result has a value equal to the extent of dimension DIM of ARRAY or, if DIM is absent, the total number of elements of ARRAY.

37       **Examples.** The value of SIZE (A (2:5, -1:1), DIM=2) is 3. The value of SIZE (A (2:5, -1:1) ) is 12.

1 **13.13.100 SPACING (X).**

2 **Description.** Returns the absolute spacing of model numbers near the argument value.

3 **Class.** Elemental function.

4 **Argument.** X must be of type real.

5 **Result Type and Type Parameter.** Same as X.

6 **Result Value.** The result has the value  $b^{e-p}$ , where  $b$ ,  $e$ , and  $p$  are as defined in 13.7.1 for the  
7 model representation of X, provided this result is within range; otherwise, the result is the  
8 same as that of TINY (X).

9 **Example.** SPACING (3.0) has the value  $2^{-22}$  for reals whose model is as at the end of 13.7.1.

10 **13.13.101 SPREAD (SOURCE, DIM, NCOPIES).**

11 **Description.** Replicates an array by adding a dimension. Broadcasts several copies of  
12 SOURCE along a specified dimension (as in forming a book from copies of a single page) and  
13 thus forms an array of rank one greater.

14 **Class.** Transformational function.

15 **Arguments.**

16 SOURCE may be of any type. It may be scalar or array valued. The rank of  
17 SOURCE must be less than 7.

18 DIM must be scalar and of type integer with value in the range  $1 \leq \text{DIM} \leq n+1$ ,  
19 where  $n$  is the rank of SOURCE.

20 NCOPIES must be scalar and of type integer.

21 **Result Type, Type Parameter, and Shape.** The result is an array of the same type and type  
22 parameters as SOURCE and of rank  $n+1$ , where  $n$  is the rank of SOURCE.

23 **Case (i):** If SOURCE is scalar, the shape of the result is (MAX (NCOPIES, 0)).

24 **Case (ii):** If SOURCE is array valued with shape  $(d_1, d_2, \dots, d_n)$ , the shape of the result is  
25  $(d_1, d_2, \dots, d_{\text{DIM}-1}, \text{MAX}(\text{NCOPIES}, 0), d_{\text{DIM}}, \dots, d_n)$ .

26 **Result Value.**

27 **Case (i):** If SOURCE is scalar, each element of the result has a value equal to SOURCE.

28 **Case (ii):** If SOURCE is array valued, the element of the result with subscripts  $(r_1, r_2, \dots,$   
29  $r_{n+1})$  has the value SOURCE  $(r_1, r_2, \dots, r_{\text{DIM}-1}, r_{\text{DIM}+1}, \dots, r_{n+1})$ .

30 **Example.** If A is the array [2, 3, 4], SPREAD (A, DIM=1, NCOPIES=NC) is the array  $\begin{bmatrix} 2 & 3 & 4 \\ 2 & 3 & 4 \\ 2 & 3 & 4 \end{bmatrix}$

31 if NC has the value 3 and is a zero-sized array if NC has the value 0.

32 **13.13.102 SQRT (X).**

33 **Description.** Square root.

34 **Class.** Elemental function.

35 **Argument.** X must be of type real or complex. Unless X is complex, its value must be greater  
36 than or equal to zero.

37 **Result Type and Type Parameter.** Same as X.

38 **Result Value.** The result has a value equal to a processor-dependent approximation to the  
39 square root of X. A result of type complex is the principal value with the real part greater  
40 than or equal to zero. When the real part of the result is zero, the imaginary part is greater

1 than or equal to zero.

2 Example. SQRT (4.0) has the value 2.0 (approximately).

3 **13.13.103 SUM (ARRAY, DIM, MASK).**

4 Optional Arguments. DIM, MASK

*corresponding to true elements of*

5 Description. Sum all the elements of ARRAY along dimension DIM ~~with mask~~ MASK.

6 Class. Transformational function.

7 Arguments.

8 ARRAY must be of type integer, real, or complex. It must not be scalar.

9 DIM (optional) must be scalar and of type integer with a value in the range  $1 \leq \text{DIM} \leq n$ ,  
10 where  $n$  is the rank of ARRAY.

11 MASK (optional) must be of type logical and must be conformable with ARRAY.

12 Result Type, Type Parameter, and Shape. The result is of the same type and type parameter  
13 as ARRAY. It is scalar if DIM is absent or ARRAY has rank one; otherwise, the result is an  
14 array of rank  $n-1$  and of shape  $(d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n)$  where  $(d_1, d_2, \dots, d_n)$  is the  
15 shape of ARRAY.

16 Result Value.

17 Case (i): The result of SUM (ARRAY) has a value equal to a processor-dependent  
18 approximation to the sum of all the elements of ARRAY or has the value zero if  
19 ARRAY has size zero.

20 Case (ii): The result of SUM (ARRAY, MASK = MASK) has a value equal to a processor-  
21 dependent approximation to the sum of the elements of ARRAY corresponding  
22 to the true elements of MASK or has the value zero if there are no true elements.

23 Case (iii): If ARRAY has rank one, SUM (ARRAY, DIM [,MASK]) has a value equal to that  
24 of SUM (ARRAY [,MASK = MASK]). Otherwise, the value of element  $(s_1, s_2, \dots,$   
25  $s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$  of SUM (ARRAY, DIM [,MASK]) is equal to SUM (ARRAY  
26  $(s_1, s_2, \dots, s_{\text{DIM}-1}, \dots, s_{\text{DIM}+1}, \dots, s_n)$  [, MASK = MASK  $(s_1, s_2, \dots, s_{\text{DIM}-1}, \dots,$   
27  $s_n)$ ]).

28 Examples.

29 Case (i): The value of SUM ((/ 1, 2, 3 /)) is 6.

30 Case (ii): SUM (C, MASK= C.GT. 0.0) forms the arithmetic sum of the positive elements  
31 of C.

32 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  
33 [9, 12].

34 **13.13.104 SYSTEM\_CLOCK (COUNT, COUNT\_RATE, COUNT\_MAX).**

35 Optional Arguments. COUNT, COUNT\_RATE, COUNT\_MAX

36 Description. Returns integer data from a real-time clock.

37 Class. Subroutine.

38 Arguments.

39 COUNT (optional) must be scalar and of type default integer. It is an INTENT (OUT) argu-  
40 ment. It is set to a processor-dependent value based on the current  
41 value of the ~~basic~~ clock or to -HUGE (0) if there is no clock. The  
42 processor-dependent value is incremented by one for each clock count

*processor*

1 until the value COUNT\_MAX is reached and is reset to zero at the next  
 2 count. It lies in the range 0 to COUNT\_MAX if there is a clock.

3 COUNT\_RATE (optional)  
 4 must be scalar and of type default integer. It is an INTENT (OUT) argu-  
 5 ment. It is set to the number of ~~basic~~ clock counts per second, or to zero  
 6 if there is no clock. *processor*

7 COUNT\_MAX (optional)  
 8 must be scalar and of type default integer. It is an INTENT (OUT) argu-  
 9 ment. It is set to the maximum value that COUNT can have, or to zero  
 10 if there is no clock.

11 **Example.** If the ~~basic system~~ *processor* clock is a 24-hour clock that registers time in 1-second intervals,  
 12 at 11:30 A.M. the reference

13 CALL SYSTEM\_CLOCK (COUNT = C, COUNT\_RATE = R, COUNT\_MAX = M)  
 14 sets  $C = 11 \times 3600 + 30 \times 60 = 41400$ ,  $R = 1$ , and  $M = 24 \times 3600 - 1 = 86399$ .

15 **13.13.105 TAN (X).**

16 **Description.** Tangent function.

17 **Class.** Elemental function.

18 **Argument.** X must be of type real.

19 **Result Type and Type Parameter.** Same as X.

20 **Result Value.** The result has a value equal to a processor-dependent approximation to  
 21  $\tan(X)$ , with X regarded as a value in radians.

22 **Example.** TAN (1.0) has the value 1.5574077 (approximately).

23 **13.13.106 TANH (X).**

24 **Description.** Hyperbolic tangent function.

25 **Class.** Elemental function.

26 **Argument.** X must be of type real.

27 **Result Type and Type Parameter.** Same as X.

28 **Result Value.** The result has a value equal to a processor-dependent approximation to  
 29  $\tanh(X)$ .

30 **Example.** TANH (1.0) has the value 0.76159416 (approximately).

31 **13.13.107 TINY (X).**

32 **Description.** Returns the smallest positive number in the model representing numbers of the  
 33 same type and type parameter as the argument.

34 **Class.** Inquiry function.

35 **Argument.** X must be of type real. It may be scalar or array valued.

36 **Result Type, Type Parameter, and Shape.** Scalar with the same type and type parameter as  
 37 X.

38 **Result Value.** The result has the value  $b^{e_{\min}-1}$  where  $b$  and  $e_{\min}$  are as defined in 13.7.1 for the  
 39 model representing numbers of the same type and type parameters as X.

40 **Example.** TINY (X) has the value  $2^{-127}$  for real X whose model is as at the end of 13.7.1.

1 **13.13.108 TRANSFER (SOURCE, MOLD, SIZE).**

2 **Optional Argument. SIZE**

3 **Description.** Returns a result with a physical representation identical to that of SOURCE but  
4 interpreted with the type and type parameters of MOLD.

5 **Class.** Transformational function.

6 **Arguments.**

7 SOURCE may be of any type and may be scalar or array valued.

8 MOLD may be of any type and may be scalar or array valued.

9 SIZE (optional) must be scalar and of type integer.

10 **Result Type, Type Parameter, and Shape.** The result is of the same type and type parame-  
11 ters as MOLD.

12 *Case (i):* If MOLD is a scalar and SIZE is absent, the result is a scalar.

13 *Case (ii):* If MOLD is array valued and SIZE is absent, the result is array valued and of  
14 rank one. Its size is as small as possible such that its physical representation is  
15 not shorter than that of SOURCE.

16 *Case (iii):* If SIZE is present, the result is array valued of rank one and size SIZE.

17 **Result Value.** If the physical representation of the result has the same length as that of  
18 SOURCE, the physical representation of the result is that of SOURCE. If the physical repre-  
19 sentation of the result is longer than that of SOURCE, the physical representation of the lead-  
20 ing part is that of SOURCE and the remainder is undefined. If the physical representation of  
21 the result is shorter than that of SOURCE, the physical representation of the result is the lead-  
22 ing part of SOURCE. If D and E are scalar variables such that the physical representation of  
23 D is as long as or longer than that of E, the value of TRANSFER (TRANSFER (E, D), E) must  
24 be the value of E. IF D is an array and E is an array of rank one, the value of TRANSFER  
25 (TRANSFER (E, D), E, SIZE (E)) must be the value of E.

26 **Examples.**

27 *Case (i):* TRANSFER (1082130432, 0.0) has the value 4.0 on a processor that represents the  
28 values 4.0 and 1082130432 as the string of binary digits 0100 0000 1000 0000 0000  
29 0000 0000 0000.

30 *Case (ii):* TRANSFER ((/ 1.1, 2.2, 3.3 /), (/ (0.0, 0.0) /)) is a complex rank-one array of  
31 length two whose first element has the value (1.1, 2.2) and whose second ele-  
32 ment has a real part with the value 3.3. The imaginary part of the second ele-  
33 ment is undefined.

34 *Case (iii):* TRANSFER ((/ 1.1, 2.2, 3.3 /), (/ (0.0, 0.0) /), 1) has the value ~~1.1 + 2.2i~~

(/ (6.1, 2.2) /)

35 **13.13.109 TRANSPOSE (MATRIX).**

36 **Description.** Transpose an array of rank two.

37 **Class.** Transformational function.

38 **Argument.** MATRIX may be of any type and must have rank two.

39 **Result Type, Type Parameters, and Shape.** The result is an array of the same type and type  
40 parameters as MATRIX and with rank two and shape (n, m) where (m, n) is the shape of  
41 MATRIX.

42 **Result Value.** Element (i, j) of the result has the value MATRIX (j, i), i = 1, 2, ..., n; j =  
43 1, 2, ..., m.

44 **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  $\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$ .



1 **13.13.110 TRIM (STRING).**2 **Description.** Returns the argument with trailing blank characters removed.3 **Class.** Transformational function.4 **Argument.** STRING must be of type character and must be a scalar.5 **Result Type and Type Parameters.** Character with the same kind type parameter value of  
6 STRING and with a length that is the length of STRING less the number of trailing blanks in  
7 STRING.8 **Result Value.** The value of the result is the same as STRING except any trailing blanks are  
9 removed. If STRING contains no nonblank characters, the result has zero length.10 **Example.** TRIM (' A B ') has the value ' A B'.11 **13.13.111 UBOUND (ARRAY, DIM).**12 **Optional Argument.** DIM13 **Description.** Returns all the upper bounds of an array or a specified upper bound.14 **Class.** Inquiry function.15 **Arguments.**16 **ARRAY** may be of any type. It must not be scalar. It must not be a pointer that  
17 is disassociated or an allocatable array that is not allocated. If ARRAY is  
18 an assumed-size array, DIM must be present with a value less than the  
19 rank of ARRAY.20 **DIM (optional)** must be scalar and of type integer with a value in the range  $1 \leq \text{DIM} \leq n$ ,  
21 where  $n$  is the rank of ARRAY.22 **Result Type, Type Parameter, and Shape.** The result is of type default integer. It is scalar if  
23 DIM is present; otherwise, the result is an array of rank one and size  $n$ , where  $n$  is the rank of  
24 ARRAY.25 **Result Value.**26 **Case (i):** UBOUND (ARRAY, DIM) has a value equal to the upper bound for subscript  
27 DIM of ARRAY if dimension DIM of ARRAY does not have size zero and has  
28 the value zero if dimension DIM has size zero. For an array section or an array  
29 expression, its value is the number of elements in the corresponding dimension.30 **Case (ii):** UBOUND (ARRAY) has a value whose  $i$ th component is equal to UBOUND  
31 (ARRAY,  $i$ ), for  $i = 1, 2, \dots, n$ , where  $n$  is the rank of ARRAY.32 **Example.** If A is declared by the statement33 `REAL A (2:3, 7:10)`

34 then UBOUND (A) is [3, 10] and UBOUND (A, DIM = 2) is 10.

35 **13.13.112 UNPACK (VECTOR, MASK, FIELD).**36 **Description.** Unpack an array of rank one into an array under the control of a mask.37 **Class.** Transformational function.38 **Arguments.**39 **VECTOR** may be of any type. It must have rank one. Its size must be at least  $t$   
40 where  $t$  is the number of true elements in MASK.41 **MASK** must be array valued and of type logical.42 **FIELD** must be of the same type and type parameters as VECTOR and must be  
43 conformable with MASK.

## Intrinsic Procedures

1 **Result Type, Type Parameter, and Shape.** The result is an array of the same type and type  
2 parameters as VECTOR and the same shape as MASK.

3 **Result Value.** The element of the result that corresponds to the  $i$ th true element of MASK, in  
4 array element order, has the value VECTOR ( $i$ ) for  $i = 1, 2, \dots, t$ , where  $t$  is the number of true  
5 values in MASK. Each other element has a value equal to FIELD if FIELD is scalar or to the  
6 corresponding element of FIELD if it is an array.

7 **Example.** Specific values may be "scattered" to specific positions in an array by using

8 UNPACK. If M is the array  $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ , V is the array [1,2,3], and Q is the logical mask

9  $\begin{bmatrix} . & T & . \\ T & . & . \\ . & . & T \end{bmatrix}$ , where "T" represents true and "." represents false, then the result of UNPACK (V,

10 MASK = Q, FIELD = M) has the value  $\begin{bmatrix} 1 & 2 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 3 \end{bmatrix}$  and the result of UNPACK (V, MASK = Q,

11 FIELD = 0) has the value  $\begin{bmatrix} 0 & 2 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 3 \end{bmatrix}$ .

### 12 13.13.113 VERIFY (STRING, SET, BACK).

13 **Optional Argument.** BACK

14 **Description.** Verify that a set of characters contains all the characters in a string <sup>by identifying</sup>  
15 <sup>the position of the first character in a string of characters that does not</sup>  
**Class.** Elemental function. <sup>appear in a given set of characters.</sup>

16 **Arguments.**

17 STRING must be of type character.

18 SET must be of type character with the same kind type parameter as  
19 STRING.

20 BACK (optional) must be of type logical.

21 **Result Type and Type Parameter.** Default integer.

22 **Result Value.**

23 **Case (i):** If BACK is absent or <sup>not</sup> present with the value false and if STRING contains at least  
24 one character that is in SET, the value of the result is the position of the leftmost  
25 character of STRING that is not in SET.

26 **Case (ii):** If BACK is <sup>not</sup> present with the value true and if STRING contains at least one char-  
27 acter that is in SET, the value of the result is the position of the rightmost charac-  
28 ter of STRING that is not in SET.

29 **Case (iii):** The value of the result is zero if each character in STRING is in SET or if  
30 STRING has zero length.

31 **Examples.**

32 **Case (i):** VERIFY ('ABBA', 'A') has the value 2.

33 **Case (ii):** VERIFY ('ABBA', 'A', BACK = .TRUE.) has the value 3.

34 **Case (iii):** VERIFY ('ABBA', 'AB') has the value 0.

## 14. SCOPE, ASSOCIATION, AND DEFINITION

1 ~~A data entity that is a constant, a variable, or a subobject of a constant.~~ Entities are identified by lexical tokens within a scope that is an executable program, a scoping unit, a single statement, or part of a statement. If the scope is an executable program, the entity is called a global entity. If the scope is a scoping unit (2.2), the entity is called a local entity. If the scope is a statement or part of a statement, the entity is called a statement entity.

6 An entity may be identified by

- 7 (1) A name (14.1),
- 8 (2) A label (14.2),
- 9 (3) An external input/output unit number (14.3),
- 10 (4) An operator symbol (14.4), or
- 11 (5) An assignment symbol (14.5).

12 By means of association, an entity may be referred to by the same identifier or a different identifier in a different scoping unit, or by a different identifier in the same scoping unit.

14 **14.1 Scope of Names.** Named entities are global, local, or statement entities.

15 **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.

18 **14.1.2 Local Entities.** Within a scoping unit, entities in the following classes:

- 19 (1) Named variables that are not statement entities (14.1.3), named constants, named constructs, statement functions, internal procedures, module procedures, dummy procedures, intrinsic procedures, generic procedures, derived types, and namelist group names,
- 23 (2) Type components, in a separate class for each type, and
- 24 (3) Argument keywords, in a separate class for each procedure with an explicit interface

25 are local entities of that scoping unit.

26 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).

29 Within a scoping unit, a name that identifies a local entity of one class must not be used to identify another local entity of the same class, except in the case of generic procedures (12.3.2.1). A name that identifies a local entity of one class may be used to identify a local entity of another class. #11-3-64!

32 Note that an intrinsic procedure is inaccessible in a scoping unit containing another local entity of the same class and having the same name. For example, in the program fragment

```
34 SUBROUTINE SUB
35     ...
36     A = SIN (K)
37     ...
38 CONTAINS
39     FUNCTION SIN (X)
40         ...
41     END FUNCTION SIN
42 END SUBROUTINE SUB
```

43 any reference to function SIN in subroutine SUB refers to the internal function SIN, not to the intrinsic function of the same name.

## Scope, Association, and Definition

- 1 The name of a local entity identifies that entity in a scoping unit and may be used to identify any
- 2 local or global entity in another scoping unit.

3 **14.1.2.1 Common Blocks.** A common block name in a scoping unit also may be the name of any  
4 local entity other than a named constant, intrinsic procedure, or a local variable that is also an  
5 external function in a function subprogram. If a name is used for both a common block and a local  
6 entity, the appearance of that name in any context other than as a common block name in a COM-  
7 MON or SAVE statement identifies only the local entity. Note that an intrinsic procedure name  
8 may be a common block name in a scoping unit that does not reference the intrinsic procedure.

9 **14.1.2.2 Function Results.** For each FUNCTION statement or ENTRY statement in a function  
10 subprogram, there is a result variable. If there is no RESULT clause, the result variable has the  
11 same name as the function being defined; otherwise, the result variable has the name specified in  
12 the RESULT clause.

13 **14.1.2.3 Unambiguous Generic Procedure References.** This subsection contains the rules that  
14 must be satisfied by every pair of specific procedures that have the same generic name, have the  
15 same generic operator, or both define assignment. They ensure that a generic reference is unam-  
16 biguous. When an intrinsic procedure, operator, or assignment is extended, the rules apply as if  
17 the intrinsic consisted of a collection of specific procedures, one for each allowed combination of  
18 type, kind type parameter, and rank for each argument or operand. When a generic procedure is  
19 accessed from a module, the rules apply to all the specific versions even if some of them are inac-  
20 cessible by their specific names.

21 Within a scoping unit, if two procedures have the same generic operator or both define assign-  
22 ment, one must have a dummy argument that corresponds by position in the argument list to a  
23 dummy argument of the other that has a different type, different kind type parameter, or different  
24 rank.

25 Within a scoping unit, two procedures that have the same name must both be subroutines or both  
26 be functions, and at least one of them must have a nonoptional dummy argument that

- are accessed by*
- 27 (1) Corresponds by position in the argument list to a dummy argument not present in the  
28 other, present with a different type, present with a different kind type parameter, or  
29 present with a different rank ~~when each is a pointer, an assumed shape array, or a scalar,~~  
30 and
  - 31 (2) Corresponds by argument keyword to a dummy argument not present in the other,  
32 present with a different type, present with a different kind type parameter, or present  
33 with a different rank ~~when each is a pointer, an assumed shape array, or a scalar.~~

34 For example, the procedures with interface bodies given by the interface block

```
35 INTERFACE A
36   SUBROUTINE AR (X)
37     REAL X
38   END SUBROUTINE AR

39   SUBROUTINE AI (J)
40     INTEGER J
41   END SUBROUTINE AI
42 END INTERFACE
```

43 satisfy rules (1) and (2). However, if J were declared REAL, rule (1) would not be satisfied while  
44 rule (2) remains satisfied; in this case, the reference to A in the statement

45 CALL A (0.0)

46 would be ambiguous.

1 **14.1.2.4 Resolving Procedure References.** The rules for interpreting a procedure reference  
2 depend on whether the procedure name in the reference is established by the available declara-  
3 tions and specifications to be generic in the scoping unit containing the reference, is established to  
4 be only specific in the scoping unit containing the reference, or is not established.

- 5 (1) A procedure name is established to be generic in a scoping unit:
- 6 (a) if that scoping unit contains an interface block with that name;
  - 7 (b) if that scoping unit contains an INTRINSIC statement with that name and it is the  
8 name of a generic intrinsic procedure;
  - 9 (c) if that scoping unit contains a USE statement that makes that procedure name  
10 accessible and the corresponding name in the module is established to be generic;  
11 or
  - 12 (d) if that scoping unit contains no declarations of that name, that scoping unit is con-  
13 tained in a host scoping unit, and that name is established to be generic in the host  
14 scoping unit.
- 15 (2) A procedure name is established to be only specific in a scoping unit if it is established  
16 to be specific and not established to be generic. It is established to be specific:
- 17 (a) if that scoping unit contains an interface body with that name;
  - 18 (b) if that scoping unit contains a module procedure, internal procedure, or statement  
19 function with that name;
  - 20 (c) if that scoping unit contains an INTRINSIC statement with that name and if it is  
21 the name of a specific intrinsic procedure;
  - 22 (d) if that scoping unit contains an EXTERNAL statement with that name;
  - 23 (e) if that scoping unit contains a USE statement that makes that procedure name  
24 accessible and the corresponding name in the module is established to be specific;  
25 or
  - 26 (f) if that scoping unit contains no declarations of that name, that scoping unit is con-  
27 tained in a host scoping unit, and that name is established to be specific in the host  
28 scoping unit.
- 29 (3) A procedure is not established in a scoping unit if it is neither established to be generic  
30 nor established to be specific.

31 **14.1.2.4.1 Resolving Procedure References to Names Established to be Generic.**

- 32 (1) If the reference is consistent with one of the specific interfaces of an interface block that  
33 has that name and either is contained in the scoping unit in which the reference appears  
34 or is made accessible by a USE statement contained in the scoping unit, the reference is  
35 to the specific procedure in that interface block that provides that interface. Note that  
36 the rules in 14.1.2.3 ensure that there can be at most one such specific procedure.
- 37 (2) If (1) does not apply, if the scoping unit contains either an INTRINSIC statement with  
38 that name or a USE statement that makes that name accessible from a module in which  
39 the corresponding name is contained in an INTRINSIC statement, and if the reference is  
40 consistent with the interface of that intrinsic procedure, the reference is to that intrinsic  
41 procedure. Note that, in the USE statement case, it is possible, because of the renaming  
42 facility, for the name in the reference to be different from the name of the intrinsic pro-  
43 cedure.
- 44 (3) If (1) and (2) do not apply, if the scoping unit is contained in a host scoping unit, if the  
45 name is established to be generic in that host scoping unit, and if there is agreement  
46 between the scoping unit and the host scoping unit as to whether the name is a function  
47 name or a subroutine name, the name is resolved by applying the rules in this section to  
48 the host scoping unit.

## Scope, Association, and Definition

- 1 (4) If (1), (2), and (3) do not apply, the procedure name must be the name of a generic  
2 intrinsic procedure, the reference must be consistent with the interface of that intrinsic  
3 procedure, and the reference is to that intrinsic procedure.

### 4 14.1.2.4.2 Resolving Procedure References to Names Established to be Only Specific.

- 5 (1) If the scoping unit contains an interface body or EXTERNAL statement with the name,  
6 if the scoping unit is a subprogram, and if the name is the name of a dummy argument  
7 of that subprogram, the dummy is a dummy procedure and the reference is to that  
8 dummy procedure. That is, the procedure invoked by executing that reference is the  
9 procedure supplied as the actual argument corresponding to that dummy procedure.
- 10 (2) If the scoping unit contains an interface body or EXTERNAL statement with the name  
11 and if (1) does not apply, the reference is to an external procedure with that name.
- 12 (3) If the scoping unit contains a module subprogram, internal subprogram, or statement  
13 function with the name, the reference is to the procedure so defined.
- 14 (4) If the scoping unit contains an INTRINSIC statement with the name, the reference is to  
15 the intrinsic with that name.
- 16 (5) If the scoping unit contains a USE statement that makes a procedure accessible by the  
17 name, the reference is to that procedure. Note that because of the renaming facility of  
18 the USE statement, the name in the reference may be different from the original name of  
19 the procedure.
- 20 (6) If none of the above apply, the scoping unit must be contained in a host scoping unit,  
21 and the reference is resolved by applying the rules in this section to the host scoping  
22 unit.

### 23 14.1.2.4.3 Resolving Procedure References to Names Not Established.

- 24 (1) If the scoping unit is a subprogram and if the name is the name of a dummy argument  
25 of that subprogram, the dummy argument is a dummy procedure and the reference is  
26 to that dummy procedure. That is, the procedure invoked by executing that reference is  
27 the procedure supplied is the actual argument corresponding to that dummy proce-  
28 dure.
- 29 (2) If (1) does not apply, if the name is the name of an intrinsic procedure, and if there is  
30 agreement between the reference and the status of the intrinsic procedure as being a  
31 function or subroutine, the reference is to that intrinsic procedure.
- 32 (3) If (1) and (2) do not apply, the reference is to an external procedure with that name.

33 **14.1.2.5 Components.** A component name has the same scope as the type of which it is a compo-  
34 nent. It may appear only within a designator of a component of a structure of that type. If the  
35 type is accessible in another scoping unit by use association or host association (14.6.1.2) and the  
36 definition of the type does not contain the PRIVATE statement (4.4.1), the component name is  
37 accessible for names of components of structures of that type in that scoping unit.

38 **14.1.2.6 Argument Keywords.** A dummy argument name in an internal procedure, module proce-  
39 dure, or a procedure interface block has a scope as an argument keyword of the scoping unit of  
40 its host. As an argument keyword, it may appear only in a procedure reference for the procedure  
41 of which it is a dummy argument. If the procedure or procedure interface block is accessible in  
42 another scoping unit by use association or host association (14.6.1.2), the argument keyword is  
43 accessible for procedure references for that procedure in that scoping unit.

44 **14.1.3 Statement Entities.** The name of a variable that appears as a dummy argument in a state-  
45 ment function statement has a scope of the statement in which it appears. It has the type and type  
46 parameters that it would have if it were the name of a variable in the scoping unit that includes the  
47 statement function.

1 The name of a variable that appears as the DO variable of an implied-DO in a DATA statement or  
2 an array constructor has a scope of the implied-DO list. It has the type and type parameter that it  
3 would have if it were the name of a variable in the scoping unit that includes the DATA statement  
4 or array constructor and this type must be integer.

5 The name of a statement entity also may be the name of a global or local entity in the same scoping  
6 unit; in this case, the name is interpreted within the statement as that of the statement entity.

7 **14.2 Scope of Labels.** A label is a local entity. No two statements in the same scoping unit  
8 may have the same label.

9 **14.3 Scope of External Input/Output Units.** An external input/output unit is a global  
10 entity.

11 **14.4 Scope of Operators.** <sup>is a local entity</sup> The intrinsic operators <sup>are global entities</sup> ~~have a scope of an executable program.~~ A  
12 defined operator ~~has a scope of a scoping unit.~~ Within a scoping unit, two operations may be iden-  
13 tified by the same operator provided they have at least one corresponding operand with different  
14 ~~type, different type parameters where neither is an asterisk, or different rank.~~ <sup>an operator may</sup>  
~~identify additional operations as specified by the rules for generic operators.~~

15 **14.5 Scope of the Assignment Symbol.** <sup>The symbol is a local entity</sup> ~~Intrinsic assignment has a scope of an executable~~  
16 ~~program. A defined assignment has a scope of a scoping unit.~~ Within a scoping unit, two assign-  
17 ~~ments may be identified by the assignment symbol provided they have at least one corresponding~~  
18 ~~operand with different type, different type parameters where neither is an asterisk, or different~~  
19 ~~rank.~~ <sup>the assignment symbol may identify additional assignment operations as</sup>  
~~specified by the rules for generic assignment.~~

20 **14.6 Association.** Two entities may become associated by name association, pointer associa-  
21 tion, or storage association.

22 **14.6.1 Name Association.** There are three forms of name association: argument association, use  
23 association, and host association. Argument, use, and host association provide mechanisms by  
24 which entities known in one scoping unit may be accessed in another scoping unit.

25 **14.6.1.1 Argument Association.** The rules governing argument association are given in Section  
26 12. As explained in 12.4, execution of a procedure reference establishes an association between an  
27 actual argument and its corresponding dummy argument. Argument association may be sequence  
28 association (12.4.1.4).

29 The name of the dummy argument may be different from the name, if any, of its associated actual  
30 argument. (Note that an actual argument may be a nameless data entity, such as an expression that  
31 is not simply a variable or constant.) The dummy argument name is the name by which the associ-  
32 ated actual argument is known, and by which it may be accessed, in the referenced procedure.

33 Upon termination of execution of a procedure reference, all argument associations established by  
34 that reference are terminated. A dummy argument of that procedure may be associated with an  
35 entirely different actual argument in a subsequent ~~execution~~ <sup>invocation</sup> of the procedure.

36 **14.6.1.2 Use Association and Host Association.** Use association is the association of names in  
37 different scoping units specified by a USE statement. The rules for use association are given in  
38 11.3.2. They allow for the renaming of the entities being accessed.

39 The rules for host association are given in 12.1.2.2.1.

40 Use association or host association allows access in one scoping unit to entities defined in another  
41 scoping unit and remains in effect throughout the execution of the executable program.

## Scope, Association, and Definition

1 **14.6.2 Pointer Association.** Pointer association between a pointer and a target allows the target  
2 to be referenced by a reference to the pointer. At different times during the execution of a pro-  
3 gram, a pointer may be undefined, associated with different targets, or be disassociated. The ini-  
4 tial association status of a pointer is undefined. If a pointer is associated with a target, the defini-  
5 tion status of the pointer is either defined or undefined, depending on the definition status of the  
6 target.

7 **14.6.2.1 Pointer Association Status.** The pointer association status of a pointer is one of follow-  
8 ing:

- 9 (1) Associated: a pointer becomes associated when
  - 10 (a) the pointer is allocated (6.3.1) as the result of the successful execution of an ALLO-  
11 CATE statement referencing the pointer, or
  - 12 (b) the pointer is pointer-assigned to a target (7.5.2) that is associated or is specified  
13 with the TARGET attribute and, if allocatable, is currently allocated.
- 14 (2) Disassociated: a pointer becomes disassociated when
  - 15 (a) the pointer is nullified (6.3.2), or
  - 16 (b) the pointer is deallocated (6.3.3), or
  - 17 (c) the pointer is pointer-assigned to a disassociated pointer (7.5.2).
- 18 (3) Undefined: the pointer association status of a pointer is undefined
  - 19 (a) initially (that is, when the pointer has never been associated or disassociated), or
  - 20 (b) if its target was never allocated, or
  - 21 (c) if its target is deallocated other than through the pointer, or
  - 22 (d) if execution of a RETURN or END statement causes the pointer's target to become  
23 undefined (6.3.3.2, item (4) of 14.7.6).

24 **14.6.2.2 Pointer Definition Status.** The definition status of a pointer is that of its target. If a  
25 pointer is associated with a definable target, the definition status of the pointer may be defined or  
26 undefined according to the rules for a variable (14.7).

27 **14.6.2.3 Relationship Between Association Status and Definition Status.** ~~An associated~~  
28 ~~pointer may be referenced and may or may not be defined.~~ If the association status of a pointer is  
29 disassociated or undefined, the pointer must not be referenced or deallocated. Whatever its associ-  
30 ation status, a pointer always may be nullified, allocated, or pointer assigned. A nullified pointer  
31 is disassociated. When a pointer is allocated, it becomes associated but undefined. When a pointer  
32 is pointer assigned, its association and definition status are determined by its target.

33 **14.6.3 Storage Association.** Storage sequences are used to describe relationships that exist  
34 among variables, common blocks, and result variables. Storage association is the association of  
35 two or more data objects that occurs when two or more storage sequences share or are aligned  
36 with one or more storage units.

37 **14.6.3.1 Storage Sequence.** A storage sequence is a sequence of storage units. The size of a  
38 storage sequence is the number of storage units in the storage sequence. A storage unit is a char-  
39 acter storage unit, a numeric storage unit, or an unspecified storage unit.

40 In a storage association context:

- 41 (1) A nonpointer scalar object of type default integer, default real, or default logical occu-  
42 pies a single numeric storage unit.
- 43 (2) A nonpointer scalar object of type double precision real or default complex occupies  
44 two contiguous numeric storage units.



- 1 (3) A nonpointer scalar object of type default character and character length one occupies  
2 one character storage unit.
- 3 (4) A nonpointer scalar object of type default character and character length *len* occupies  
4 *len* contiguous character storage units.
- 5 (5) A nonpointer scalar object of type nondefault integer, real other than default or double  
6 precision, nondefault logical, nondefault complex, nondefault character of any length,  
7 or nonsequence type occupies a single unspecified storage unit that is different for each  
8 case.
- 9 (6) A nonpointer array of intrinsic type or sequence derived type occupies a sequence of  
10 contiguous storage sequences, one for each array element, in array element order  
11 (6.2.2.2).
- 12 (7) A nonpointer scalar object of sequence type occupies a sequence of storage sequences  
13 corresponding to the sequence of its ultimate components.
- 14 (8) A pointer occupies a single unspecified storage unit that is different from that of any  
15 nonpointer object and is different for each combination of type, type parameters, and  
16 rank.
- 17 A sequence of storage sequences forms a storage sequence. The order of the storage units in such a  
18 composite storage sequence is that of the individual storage units in each of the constituent storage  
19 sequences taken in succession, ignoring any zero-sized constituent sequences.
- 20 Each common block has a storage sequence (5.5.2.1).
- 21 **14.6.3.2 Association of Storage Sequences.** Two nonzero-sized storage sequences  $s_1$  and  $s_2$   
22 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  
23 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  
24  $k$  such that  $1 \leq i+k \leq \text{size of } s_1$  and  $1 \leq j+k \leq \text{size of } s_2$ .
- 25 Storage association also is defined between two zero-sized storage sequences, and between a zero-  
26 sized storage sequence and a storage unit. A zero-sized storage sequence in a sequence of storage  
27 sequences is storage associated with its successor, if any. If the successor is another zero-sized  
28 storage sequence, the two sequences are storage associated. If the successor is a nonzero-sized  
29 storage sequence, the zero-sized sequence is storage associated with the first storage unit of the  
30 successor. Two storage units that are each storage associated with the same zero-sized storage  
31 sequence are the same storage unit.
- 32 **14.6.3.3 Association of Scalar Data Objects.** Two scalar data objects are storage associated if  
33 their storage sequences are storage associated. Two scalar entities are **totally associated** if they  
34 have the same storage sequence. Two scalar entities are **partially associated** if they are ~~storage~~  
35 associated without being totally associated.
- 36 The definition status and value of a data object affects the definition status and <sup>value</sup> of any stor-  
37 age associated entity. An EQUIVALENCE statement, a COMMON statement, an ENTRY state-  
38 ment, ~~or a procedure reference~~ may cause storage association of storage sequences.
- 39 An EQUIVALENCE statement causes storage association of data objects only within one scoping  
40 unit, unless one of the equivalenced entities is also in a common block (5.5.1.1 and 5.5.2.1).
- 41 COMMON statements cause data objects in one scoping unit to become storage associated with  
42 data objects in another scoping unit.
- 43 A named common block is permitted to contain a sequence of differing storage units provided  
44 each scoping unit that accesses the common block specifies an identical sequence of storage units.  
45 The same rule applies to blank common blocks. If the sizes of the two blank common blocks differ,  
46 the sequence of storage units of the shorter block must be identical to the initial sequence of the  
47 storage units of the longer block.
- 48 An ENTRY statement in a function subprogram causes <sup>storage</sup> association of the result variables.

## Scope, Association, and Definition

- 1 Partial association may exist only between
- 2 (1) An object of default character or character sequence type and an object of default character or character sequence type or
- 3
- 4 (2) An object of default complex, double precision real, or numeric sequence type and an
- 5 object of default integer, default real, default logical, double precision real, default complex, or numeric sequence type.
- 6

7 For noncharacter entities, partial association may occur only through the use of COMMON, EQUIVALENCE, or ENTRY statements. For character entities, partial association may occur only through argument association or the use of COMMON, EQUIVALENCE, or ENTRY statements.

10 In the example:

```
11 REAL A (4), B
12 COMPLEX C (2)
13 DOUBLE PRECISION D
14 EQUIVALENCE (C (2), A (2), B), (A, D)
```

15 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:

```
17 Storage unit      1      2      3      4      5
18 -----C (1) -----|-----C (2) -----
19                   A (1)  A (2)  A (3)  A (4)
20                   -----B-----
21                   -----D-----
```

22 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 storage associated with D, C (1) and C (2) are not storage associated with each other.

26 Partial association of character entities occurs when some, but not all, of the storage units of the entities are the same. In the example:

```
28 CHARACTER A*4, B*4, C*3
29 EQUIVALENCE (A (2:3), B, C)
```

30 A, B, and C are partially associated.

31 **14.7 Definition and Undefined 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.

36 **14.7.1 Definition of Objects and Subobjects.** Arrays, including sections, and variables of derived, character, or complex type are objects that consist of zero or more subobjects. Associations may be established between variables and subobjects and between subobjects of different variables. These subobjects may become defined or undefined.

- 40 (1) An object is defined if and only if all of its subobjects are defined.
- 41 (2) If an object is undefined, at least one (but not necessarily all) of its subobjects are undefined.
- 42

43 **14.7.2 Variables That Are Always Defined.** Zero-sized arrays and zero-length strings are always defined.

44

1 **14.7.3 Variables That Are Initially Defined.** The following variables are initially defined:

- 2 (1) Variables specified to have initial values by DATA statements,
- 3 (2) Variables specified to have initial values by type declaration statements, and
- 4 (3) Variables that are always defined.

5 **14.7.4 Variables That Are Initially Undefined.** All other variables are initially undefined.

6 **14.7.5 Events That Cause Variables to Become Defined.** Variables become defined as follows:

- 7 (1) Execution of an intrinsic assignment statement other than a masked array assignment  
8 statement causes the variable that precedes the equals to become defined. Execution of  
9 a defined assignment statement may cause all or part of the variable that precedes the  
10 equals to become defined.
- 11 (2) Execution of a masked array assignment statement may cause some or all of the array  
12 elements in the assignment statement to become defined (7.5.3).
- 13 (3) As execution of an input statement proceeds, each variable that is assigned a value from  
14 the input file becomes defined at the time that data is transferred to it. (See (5) in  
15 14.7.6.) Execution of a WRITE statement whose unit specifier identifies an internal file  
16 causes each record that is written to become defined.
- 17 (4) Execution of a DO statement causes the DO variable, if any, to become defined.
- 18 (5) Beginning of execution of the action specified by an implied-DO list in an input/output  
19 statement causes the implied-DO variable to become defined.
- 20 (6) Execution of an ASSIGN statement causes the variable in the statement to become defined with a statement  
21 label value.
- 22 (7) A reference to a procedure causes the entire dummy argument data object to become  
23 defined if the entire corresponding actual argument is defined with a value that is not a state-  
24 ment label.  
25 A reference to a procedure causes a subobject of a dummy argument to become defined  
26 if the corresponding subobject of the corresponding actual argument is defined.
- 27 (8) Execution of an input/output statement containing an input/output IOSTAT= specifier  
28 causes the specified integer variable to become defined.
- 29 (9) Execution of a READ statement containing a SIZE= specifier causes the specified integer  
30 variable to become defined.
- 31 (10) Execution of an INQUIRE statement causes any variable that is assigned a value during  
32 the execution of the statement to become defined if no error condition exists.
- 33 (11) When a character storage unit becomes defined, all associated character storage units  
34 become defined.  
35 When a numeric storage unit becomes defined, all associated numeric storage units of  
36 the same type become defined, except that variables associated with the variable in an ASSIGN state-  
37 ment become undefined when the ASSIGN statement is executed. When an entity of double preci-  
38 sion real type becomes defined, all totally associated entities of double precision real  
39 type become defined.
- 40 (12) When an unspecified storage unit becomes defined, all associated unspecified storage  
41 units become defined.
- 42 (13) When a default complex entity becomes defined, all partially associated default real  
43 entities become defined.
- 44 (14) When both parts of a default complex entity become defined as a result of partially  
45 associated default real or default complex entities becoming defined, the default com-  
46 plex entity becomes defined.

## Scope, Association, and Definition

- 1 (14) When all components of a numeric sequence structure or character sequence structure  
2 become defined as a result of partially associated objects becoming defined, the struc-  
3 ture becomes defined.
- 4 (15) Execution of an ALLOCATE or DEALLOCATE statement with a STAT= specifier  
5 causes the variable specified by the STAT= specifier to become defined.
- 6 (16) Allocation of a zero-sized array causes the array to become defined.
- 7 (17) Invocation of a procedure causes any automatic object of zero size in that procedure to  
8 become defined.
- 9 (18) Execution of a pointer assignment statement that associates a pointer with a target that  
10 is defined causes the pointer to become defined.

11 **14.7.6 Events That Cause Variables to Become Undefined.** Variables become undefined as  
12 follows:

- 13 (1) When a variable of a given type becomes defined, all associated variables of different  
14 type become undefined. However, when a variable of type default real is partially asso-  
15 ciated with a variable of type default complex, the complex variable does not become  
16 undefined when the real variable becomes defined and the real variable does not  
17 become undefined when the complex variable becomes defined. When a variable of  
18 type default complex is partially associated with another variable of type default com-  
19 plex, definition of one does not cause the other to become undefined.
- 20 (2) Execution of an ASSIGN statement causes the variable in the statement to become undefined as an integer.  
21 Variables that are associated with the variable also become undefined.
- 22 (3) If the evaluation of a function may cause an argument of the function or a variable in a  
23 module or in a common block to become defined and if a reference to the function  
24 appears in an expression in which the value of the function is not needed to determine  
25 the value of the expression, the argument or variable becomes undefined when the  
26 expression is evaluated.
- 27 (4) The execution of a RETURN statement or an END statement within a subprogram  
28 causes all variables local to its scoping unit or local to the current instance of its scoping  
29 unit for a recursive invocation to become undefined except for the following:
  - 30 (a) Variables with the SAVE attribute.
  - 31 (b) Variables in blank common.
  - 32 (c) Variables in a named common block that appears in the subprogram and appears  
33 in at least one other scoping unit that is making either a direct or indirect reference  
34 to the subprogram.
  - 35 (d) Variables accessed from the host scoping unit.
  - 36 (e) Variables accessed from a module that also is referenced directly or indirectly by  
37 at least one other scoping unit that is making either a direct or indirect reference to  
38 the subprogram.
  - 39 (f) Variables in a named common block that are initially defined (14.7.3) and that  
40 have not been subsequently defined or redefined. *or undefined*
- 41 (5) When an error condition or end-of-file condition occurs during execution of an input  
42 statement, all of the variables specified by the input list or *namelist-group* of the state-  
43 ment become undefined.
- 44 (6) When an error or end-of-file condition occurs during execution of an input/output  
45 statement, some or all of the implied-DO variables may become undefined (9.4.3).
- 46 (7) Execution of a defined assignment statement may leave all or part of the variable that  
47 precedes the equals undefined.

*doesn't mention 12-1-46 ff*

- 1 (8) Execution of a direct access input statement that specifies a record that has not been
- 2 written previously causes all of the variables specified by the input list of the statement
- 3 to become undefined.
- 4 (9) Execution of an INQUIRE statement may cause the NAME=, RECL=, and NEXTREC=
- 5 variables to become undefined (9.6).
- 6 (10) When a character storage unit becomes undefined, all associated character storage units
- 7 become undefined.
- 8 When a numeric storage unit becomes undefined, all associated numeric storage units
- 9 become undefined unless the undefinition is a result of defining an associated numeric
- 10 storage unit of different type (see (1) above).
- 11 When an entity of double precision real type becomes undefined, all totally associated
- 12 entities of double precision real type become undefined.
- 13 When an unspecified storage unit becomes undefined, all associated unspecified storage
- 14 units become undefined.
- 15 (11) A reference to a procedure causes part of a dummy argument to become undefined if the corresponding part
- 16 of the actual argument is defined with a value that is a statement label value.
- 17 (12) When an allocatable array is deallocated, it becomes undefined. Successful execution of
- 18 an ALLOCATE statement causes the allocated array to become undefined.
- 19 (13) Execution of an INQUIRE statement causes all inquiry specifier variables to become
- 20 undefined if an error condition exists, except for the variable in the IOSTAT= specifier,
- 21 if any.
- 22 (14) When a procedure is invoked:
- 23 (a) An optional dummy argument that is not associated with an actual argument is
- 24 undefined.
- 25 (b) A dummy argument with INTENT (OUT) is undefined.
- 26 (c) An actual argument associated with a dummy argument with INTENT (OUT)
- 27 becomes undefined.
- 28 (d) A subobject of a dummy argument is undefined if the corresponding subobject of
- 29 the actual argument is undefined.
- 30 (e) The result variable of a function is undefined.
- 31 (15) When the association status of a pointer becomes undefined or disassociated (6.3), the
- 32 pointer becomes undefined.

*What causes this?*

*need "events that cause..." section*

**14.8 Allocation Status.** The allocation status of an allocatable array is one of the following at any time during the execution of an executable program:

- 35 (1) Not currently allocated, which means that the array has never been allocated or that the
- 36 last operation on it was a deallocation.
- 37 (2) Currently allocated, which means that the array has been allocated by an ALLOCATE
- 38 statement and has not been subsequently deallocated.
- 39 (3) Undefined, which means that the array does not have the SAVE attribute and was cur-
- 40 rently allocated when execution of a RETURN or END statement resulted in no execut-
- 41 ing scoping units having access to it.

42 If the allocation status of an allocatable array is currently allocated, the array may be referenced

43 and defined. An allocatable array that is not currently allocated must not be referenced or defined.

44 If the allocation status of an allocatable array is undefined, the array must not be referenced,

45 defined, allocated, or deallocated.



## ANNEX A. GLOSSARY OF TECHNICAL TERMS

- 1 (This Annex is not part of this standard, but is included for information only.)
- 2 The following is a list of the principal technical terms used in the standard and their definitions. A  
3 reference in parentheses immediately after a term is to the section where the term is defined or  
4 explained. The wording of a definition here is not necessarily the same as in the standard. Where  
5 the definition uses a term that is itself defined in this glossary, the first occurrence of the term in  
6 that definition is printed in italics.
- 7 **action statement.** A single *statement* specifying a computational action (R216).
- 8 **actual argument** (12.4.1). An *expression*, a *variable*, a *procedure*, or an alternate return specifier that  
9 is specified in a *procedure reference*.
- 10 **allocatable array** (5.1.2.4.3). A *named array* <sup>having the</sup> ~~whose type, type parameters, and rank are specified in a~~  
11 ~~type declaration statement containing an~~ ALLOCATABLE attribute. Only when it has space allocated  
12 for it does it have a *shape* and may it be *referenced* or *defined*.
- 13 **argument** (12). An *actual argument* or a *dummy argument*.
- 14 **argument association** (14.6.1.1). The relationship between an *actual argument* and a *dummy argu-*  
15 *ment* during the execution of a *procedure reference*.
- 16 **argument keyword** (2.5.2). A *dummy argument name*. It may be used in a *procedure reference* ahead  
17 of the equals symbol (R1211) provided the procedure has an *explicit interface*.
- 18 **array** (2.4.7). A set of *scalar data*, all of the same *type* and *type parameters*, whose individual ele-  
19 ments are arranged in a rectangular pattern. It may be a *named array*, the *target* of an *array pointer*,  
20 an *array section*, a *structure component*, a *function value*, or an *expression*. Its *rank* is at least one.  
21 Note that in FORTRAN 77, arrays were always named and always variables.
- 22 **array element** (2.4.7, 6.2.2.1). One of the *scalar data* that make up an *array*.
- 23 **array pointer** (5.1.2.4.3). A *pointer* to an *array*.
- 24 **array section** (6.2.2.3). A *subobject* of an *array* consisting of a set of *array elements* or *substrings* of  
25 array elements. The set is specified by *subscripts*, *subscript triplets*, and *vector subscripts*.
- 26 **array-valued.** Having the property of being an *array*.
- 27 **assignment statement** (7.5.1.1). A *statement* of the form '*variable = expression*'.
- 28 **association** (14.6). *Name association*, *pointer association*, or *storage association*.
- 29 **assumed-size array** (5.1.2.4.4). A *dummy array* whose *size* is assumed from the associated *actual*  
30 *argument*. Its last upper bound is specified by an asterisk.
- 31 **attribute** (5). A property of a *data object* that may be specified in a *type declaration statement* (R501).
- 32 **automatic data object** (5.1). A *data object* that is a *local entity* of a *subprogram*, that is not a *dummy*  
33 *argument*, and that has a nonconstant character length or array bound.
- 34 **belong** (8.1.4.4.3, 8.1.4.4.4). If an EXIT or a CYCLE *statement* contains a *construct name*, the state-  
35 ment *belongs* to the DO construct using that name. Otherwise, it *belongs* to the innermost DO  
36 construct in which it appears.
- 37 **block** (8.1). A sequence of *executable constructs* embedded in another executable construct,  
38 bounded by *statements* that are particular to the construct, and treated as an integral unit.
- 39 **block data program unit** (11.4). A *program unit* that provides initial values for *data objects* in *named*  
40 *common blocks*.
- 41 **bounds** (5.1.2.4.1). For a *named array*, the limits within which the values of the *subscripts* of its *array*  
42 *elements* must lie.
- 43 **character.** A letter, digit, or other symbol.
- 44 **characteristics** (12.2)

## Glossary of Technical Terms

- 1 (1) Of a *procedure*, its classification as a *function* or *subroutine*, the characteristics of its  
2 *dummy arguments*, and the characteristics of its *function result* if it is a function.
- 3 (2) Of a *dummy argument*, whether it is a *data object*, is a *procedure*, or has the OPTIONAL  
4 *attribute*.
- 5 (3) Of a *data object*, its *type*, *type parameters*, *shape*, the exact dependence of an array bound or  
6 the character length on other entities, *intent*, whether it is optional, whether it is a *pointer*  
7 or a *target*, and whether the *shape*, *size*, or *character length* is assumed.
- 8 (4) Of a *dummy procedure*, whether the interface is explicit, the characteristics of the proce-  
9 dure if the interface is explicit, and whether it is optional.
- 10 (5) Of a *function result*, its type, type parameters, whether it is a pointer, rank if it is a  
11 pointer, shape if it is not a pointer, the exact dependence of an array bound or the char-  
12 acter length on other entities, and whether the character length is assumed.
- 13 **character string** (4.3.2.1). A sequence of *characters* numbered from left to right 1, 2, 3, ...
- 14 **character storage unit** (14.6.3.1). The unit of storage for holding a *datum* of type *character*.  
15 *scalar that is not a pointer*  
*and is of type default character and length 1.*
- 16 **collating sequence**. An ordering of all the different *characters* of a particular *kind type parameter*.
- 17 **common block** (5.5.2). A block of physical storage that may be accessed by any of the *scoping units*  
18 in an *executable program*.
- 19 **component** (4.4). A constituent of a *derived type*.
- 20 **conformable** (2.4.7). Two *arrays* are said to be conformable if they have the same *shape*. A *scalar* is  
21 conformable with any array.
- 22 **conformance** (1.4). An *executable program* conforms to the standard if it uses only those forms and  
23 relationships described therein and if the executable program has an interpretation according to  
24 the standard. A *program unit* conforms to the standard if it can be included in an executable pro-  
25 gram in a manner that allows the executable program to be standard conforming. A *processor* con-  
26 forms to the standard if it executes standard-conforming programs in a manner that fulfills the  
27 interpretations prescribed in the standard.
- 28 **connected** (9.3.2).
- 29 (1) For an *external unit*, the property of referring to an *external file*.
- 30 (2) For an *external file*, the property of having an *external unit* that refers to it.
- 31 **constant** (2.4.4). A *data object* whose value must not change during execution of an *executable pro-*  
32 *gram*. It may be a *named constant* or a *literal constant*. ~~Note that in FORTRAN 77, constants were~~  
~~always literal constants.~~
- 33 **constant expression** (7.1.6.1). An *expression* satisfying rules that ensure that its value does not vary  
34 during program execution.
- 35 **construct** (8). A sequence of *statements* starting with a CASE, DO, IF, or WHERE statement and  
36 ending with the corresponding terminal statement.
- 37 **data**. Plural of *datum*.
- 38 **data entity** (2.4.3). A *data object*, the result of the evaluation of an *expression*, or the result of the  
39 execution of a *function reference* (called the *function result*). A data entity has a *data type* (either  
40 *intrinsic* or *derived*) and has, or may have, a data value (the exception is an *undefined variable*).  
41 Every data entity has a *rank* and is thus either a *scalar* or an *array*.
- 42 **data object** (2.4.3). A *data entity* that is a *constant*, a *variable*, or a subobject of a constant.
- 43 **data type** (2.4.1). A *named category* of *data* that is characterized by a set of values, together with a  
44 way to denote these values and a collection of *operations* that interpret and manipulate the values.  
45 For an *intrinsic type*, the set of data values depends on the values of the *type parameters*.



- 1 **datum.** A single quantity that may have any of the set of values specified for its *data type*.
- 2 **definable** (2.5.4). A *variable* is **definable** if its value may be changed by the appearance of its *name*  
3 or *designator* on the left of an *assignment statement*. An *allocatable array* that has not been allocated is  
4 an example of a *data object* that is not definable. An example of a *subobject* that is not definable is C  
5 (I) when C is an *array* that is a *constant* and I is an integer variable.
- 6 **defined** (2.5.4). For a *data object*, the property of having or being given a valid value.
- 7 **defined assignment statement** (7.5.1.3). An *assignment statement* <sup>and an interface block that</sup> ~~that is not an intrinsic assignment~~  
8 ~~statement and is defined by a subroutine that has an explicit ASSIGNMENT(=) interface.~~  
9 <sup>Specifies ASSIGNMENT(=).</sup>
- 9 **defined operation** (7.1.3). An *operation* that is not an *intrinsic operation* and is defined by a *function*  
10 that is associated with a *generic identifier*.
- 11 **deleted feature** (1.6). A feature in FORTRAN 77 that is considered to have been redundant and  
12 largely unused. No features in FORTRAN 77 have been deleted from the standard. Note that a fea-  
13 ture designated as an *obsolescent feature* in the standard may become a deleted feature in the next  
14 revision.
- 15 **derived type** (2.4.1.2, 4.4). A *type* whose *data* have *components*, each of which is either of *intrinsic*  
16 *type* or of another *derived type*.
- 17 **designator.** See *subobject designator*.
- 18 **disassociated** (2.4.8). A *pointer* is **disassociated** following execution of a DEALLOCATE or NUL-  
19 LIFY *statement*.
- 20 **dummy argument** (12.5.2.2, 12.5.2.3, 12.5.2.5, 12.5.4). An entity whose *name* appears in the paren-  
21 thesized list following the *procedure name* in a FUNCTION *statement*, a SUBROUTINE *statement*, an  
22 ENTRY *statement*, or a *statement function statement*.
- 23 **dummy array.** A *dummy argument* that is an *array*.
- 24 **dummy pointer.** A *dummy argument* that is a *pointer*.
- 25 **dummy procedure** (12.1.2.3). A *dummy argument* that is specified or *referenced* as a *procedure*.
- 26 **elemental** (12.4.3, 12.4.5). An adjective applied to an *intrinsic operation*, *procedure*, or *assignment*  
27 *statement* that is applied independently to elements of an *array* or corresponding elements of a set  
28 of *conformable arrays* and *scalars*.
- 29 **entity.** The term used for any of the following: a *program unit*, a *procedure*, an *operator*, an *interface*  
30 *block*, a *common block*, an *external unit*, a *statement function*, a *type*, a *named variable*, an *expression*, a  
31 *component* of a *structure*, a *named constant*, a *statement label*, a *construct*, or a *namelist group*.
- 32 **executable construct** (2.1). A CASE, DO, IF, or WHERE *construct* or an *action statement* (R216).
- 33 **executable program** (2.2.1). A set of *program units* that includes exactly one *main program*.
- 34 **executable statement** (2.3.1). An instruction to perform or control one or more computational  
35 actions.
- 36 **explicit interface** (12.3.1). For a *procedure referenced* in a *scoping unit*, the property of being an *inter-*  
37 *nal procedure*, a *module procedure*, an *intrinsic procedure*, an *external procedure* that has an *interface*  
38 *block*, or a *dummy procedure* that has an *interface block*.
- 39 <sup>the recursive procedure referenced in its own scoping unit</sup>  
**explicit-shape array** (5.1.2.4.1). A *named array* that is declared with *explicit bounds*.
- 40 **expression** (7.1). A sequence of *operands*, *operators*, and *parentheses* (R723). It may be a *variable*, a  
41 *constant*, a *function reference*, or may represent a *computation*.
- 42 **extent** (2.4.7). The size of one dimension of an *array*.
- 43 **external file** (9.2.1). A sequence of *records* that exists in a medium external to the *executable pro-*  
44 *gram*.
- 45 **external procedure** (2.2.3.1). A *procedure* that is defined by an *external subprogram* or by a means  
46 other than Fortran.

## Glossary of Technical Terms

- 1 **external subprogram** (2.2). A *subprogram* that is not contained in a *main program*, *module*, or  
2 another subprogram. Note that a *module* is not called a subprogram. Note that in FORTRAN 77, a  
3 *block data program unit* is called a subprogram.
- 4 **external unit** (9.3). A mechanism that is used to refer to an *external file*. It is identified by a non-  
5 negative integer.
- 6 **file** (9.2). An *internal file* or an *external file*.
- 7 **function** (2.2.3). A *procedure* that is invoked in an *expression*.
- 8 **function result** (12.5.2.2). The *data object* that returns the value of a *function*.
- 9 **function subprogram** (12.5.2.2). A sequence of *statements* beginning with a FUNCTION statement  
10 that is not in an *interface block* and ending with the corresponding END statement.
- 11 **generic identifier**. A *lexical token* that appears in an INTERFACE statement and is associated with  
12 all the *procedures* in the *interface block*.
- 13 **global entity** (14.1.1). An *entity* identified by a *lexical token* whose *scope* is an *executable program*. It  
14 may be a *program unit*, a *common block*, or an *external procedure*.
- 15 **host** (2.2.3.3). A *main program* or *subprogram* that contains an *internal procedure* is called the *host* of  
16 the *internal procedure*. A *module* that contains a *module procedure* is called the *host* of the *module*  
17 *procedure*.
- 18 **host association** (11.2.2). The process by which an *internal subprogram*, *module subprogram*, or  
19 *derived type definition* accesses *entities* of its *host*.
- 20 **implicit interface** (12.3.1). A *procedure* referenced in a *scoping unit* is said to have an *implicit inter-*  
21 *face* if the *procedure* is an *external procedure* that does not have an *interface block*, a *dummy procedure*  
22 that does not have an *interface block*, or a *statement function*.
- 23 **inquiry function**. An *intrinsic function* whose result depends on properties of the principal *argu-*  
24 *ment* other than the value of the *argument*.
- 25 **intent** (12.5.2.1). An attribute of a *dummy argument* that is neither a *procedure* nor a *pointer*, which  
26 indicates whether it is used to transfer data into the *procedure*, out of the *procedure*, or both.
- 27 **instance of a subprogram** (12.5.2.4). The copy of a *subprogram* that is created when a *procedure*  
28 defined by the *subprogram* is *invoked*.
- 29 **interface block** (12.3.2.1). A sequence of *statements* from an INTERFACE statement to the corre-  
30 sponding END INTERFACE statement.
- 31 **interface body** (12.3.2.1). A sequence of *statements* in an *interface block* from a FUNCTION or SUB-  
32 ROUTINE statement to the corresponding END statement.
- 33 **interface of a procedure** (12.3). See *procedure interface*.
- 34 **internal file** (9.2.2). A *character variable* that is used to transfer and convert *data* from internal stor-  
35 age to internal storage.
- 36 **internal procedure** (2.2.3.3). A *procedure* that is defined by an *internal subprogram*.
- 37 **internal subprogram** (2.2). A *subprogram* contained in a *main program* or another subprogram.
- 38 **intrinsic** (2.5.7). An adjective applied to *types*, *operations*, *assignment statements*, and *procedures* that  
39 are defined in the standard and may be used in any *scoping unit* without further definition or speci-  
40 fication.
- 41 **invoke** (2.2.3).
- 42 (1) To call a *subroutine* by a CALL statement or by a *defined assignment statement*.
- 43 (2) To call a *function* by a *reference* to it by *name* or *operator* during the evaluation of an  
44 *expression*.

- 1 **keyword** (2.5.2). *Statement keyword* or *argument keyword*.
- 2 **kind type parameter**. A parameter whose values label the available kinds of an *intrinsic type*.
- 3 **label**. See *statement label*.
- 4 **length of a character string** (4.3.2.1). The number of *characters* in the *character string*.
- 5 **lexical token** (3.2). A sequence of one or more characters with an indivisible interpretation.
- 6 **line**. A source-form *record* containing from 0 to 132 *characters*.
- 7 **literal constant** (2.4.4). A *constant* without a *name*.
- 8 **local entity** (14.1.2). An *entity* identified by a *lexical token* whose *scope* is a *scoping unit*.
- 9 **main program** (2.2.2, 11.1). A *program unit* that is not a *module*, *subprogram*, or *block data program unit*.
- 10
- 11 **many-one array section** (6.2.2.3.2). An *array section* with a *vector subscript* having two or more elements with the same value.
- 12
- 13 **module** (2.2.4, 11.3). A *program unit* that contains or accesses definitions to be accessed by other program units.
- 14
- 15 **module procedure** (2.2.3.2). A *procedure* that is defined by a *module subprogram*.
- 16 **module subprogram** (2.2). A *subprogram* that is contained in a *module* but is not an *internal subprogram*.
- 17
- 18 **name** (3.2.2). A *lexical token* consisting of a letter followed by up to 30 alphanumeric characters (letters, digits, and underscores). Note that in FORTRAN 77, this was called a symbolic name.
- 19
- 20 **name association** (14.6.1). *Argument association*, *use association*, or *host association*.
- 21 **named**. Having a *name*.
- 22 **named constant** (2.4.4). A *constant* that has a *name*. Note that in FORTRAN 77, this was called a symbolic constant.
- 23
- 24 **numeric storage unit** (2.4.9). The unit of storage for holding a *datum* of type default real, default integer, or default logical.
- 25
- 26 **numeric type**. Integer, real or complex *type*.
- 27 **object** (2.4.3.1). *Data object*.
- 28 **obsolescent feature** (1.6). A feature in FORTRAN 77 that is considered to have been redundant but that is still in frequent use.
- 29
- 30 **operand** (2.5.8). An *expression* that precedes or succeeds an *operator*.
- 31 **operation** (7.1.2). A computation involving one or two *operands*.
- 32 **operator** (2.5.8). A *lexical token* that specifies an *operation*.
- 33 **pointer** (2.4.8). <sup>(2.4.c) A variable that has</sup> ~~A named data object whose type, type parameters, and rank are specified in a type declaration statement containing~~ the POINTER attribute. A pointer must not be referenced or defined unless it is *pointer associated* with a *target*. If it is an array, it does not have a *shape* unless it is *pointer associated*.
- 34
- 35
- 36
- 37 **pointer assignment** (7.5.2). The *pointer association* of a *pointer* with a *target* by the execution of a *pointer assignment statement* or the execution of an *assignment statement* for a *data object* of *derived type* having the pointer as a *subobject*.
- 38
- 39
- 40 **pointer assignment statement** (7.5.2). A *statement* of the form *pointer-name => target*.
- 41 **pointer associated** (6.3, 7.5.2). The relationship between a *pointer* and a *target* following a *pointer assignment* or a valid execution of an ALLOCATE *statement*.
- 42

## Glossary of Technical Terms

- 1 **pointer association** (14.6.2). The process by which a *pointer* becomes *pointer associated* with a *target*.
- 2 **present** (12.5.2.8). A *dummy argument* is **present** in an *instance of a subprogram* if it is *associated* with  
3 an *actual argument* and the *actual argument* is a *dummy argument* that is *present* in the *invoking*  
4 *procedure* or is not a *dummy argument* of the *invoking procedure*.
- 5 **procedure** (2.2.3, 12.1). A computation that may be *invoked* during *program execution*. It may be a  
6 *function* or a *subroutine*. It may be an *intrinsic procedure*, an *external procedure*, a *module procedure*,  
7 an *internal procedure*, a *dummy procedure*, or a *statement function*. A *subprogram* may define more  
8 than one *procedure* if it contains *ENTRY statements*.
- 9 **procedure interface** (12.3). The *characteristics* of a *procedure*, the *name* of the *procedure*, the *name* of  
10 each *dummy argument*, and the *generic identifiers* (if any) by which it may be *referenced*.
- 11 **processor** (1.2). The combination of a *computing system* and the mechanism by which *executable*  
12 *programs* are transformed for use on that *computing system*.
- 13 **program**. See *executable program* and *main program*.
- 14 **program unit** (2.2). The fundamental component of an *executable program*. A sequence of *state-*  
15 *ments* and *comment lines*. It may be a *main program*, a *module*, an *external subprogram*, or a *block data*  
16 *program unit*.
- 17 **rank** (2.4.7). The number of dimensions of an *array*. Zero for a *scalar*.
- 18 **record** (9.1). A sequence of values that is treated as a whole within a *file*.
- 19 **reference** (2.5.5). The appearance of a *data object name* or *subobject designator* in a context requiring  
20 the value at that point during *execution*, or the appearance of a *procedure name*, its *operator symbol*,  
21 or a *defined assignment statement* in a context requiring *execution* of the *procedure* at that point.  
22 Note that neither the act of *defining a variable* nor the appearance of the name of a *procedure* as an  
23 *actual argument* is regarded as a *reference*.
- 24 **scalar** (2.4.6).
- 25 (1) A single *datum* that is not an *array*.
- 26 (2) Not having the property of being an *array*.
- 27 **scope** (14). That part of an *executable program* within which a *lexical token* has a single *interpreta-*  
28 *tion*. It may be an *executable program*, a *scoping unit*, a *single statement*, or a *part of a statement*.
- 29 **scoping unit** (14). One of the following:
- 30 (1) A *derived-type definition*, *derived type definitions and*
- 31 (2) An *interface body*, excluding any *interface bodies* contained within it, or
- 32 (3) A *program unit* or *subprogram*, excluding *derived-type definitions*, *interface bodies*, and  
33 *subprograms* contained within it.
- 34 **section subscript** (6.2.2). A *subscript*, *vector subscript*, or *subscript triplet* in an *array section selector*.
- 35 **selector**. A syntactic mechanism for designating
- 36 (1) Part of a *data object*. It may designate a *substring*, an *array element*, an *array section*, or a  
37 *structure component*.
- 38 (2) The set of values for which a *CASE block* is executed.
- 39 **shape** (2.4.7). For an *array*, the *rank* and *extents*. The shape may be represented by the *rank-one*  
40 *array* whose elements are the *extents* in each dimension.
- 41 **size** (2.4.7). For an *array*, the total number of elements.
- 42 **standard module** (1.7). A *module* standardized as a separate *collateral standard*.
- 43 **statement** (3.3). A sequence of *lexical tokens*. It usually consists of a single line, but the *ampersand*  
44 *symbol* may be used to continue a *statement* from one line to another and the *semicolon symbol*  
45 may be used to separate *statements* within a line.

## Glossary of Technical Terms

- 1 **statement entity** (14). An *entity* identified by a *lexical token* whose *scope* is a single *statement* or part  
2 of a *statement*.
- 3 **statement function** (12.5.4). A *procedure* specified by a single *statement* that is similar in form to an  
4 *assignment statement*.
- 5 **statement keyword** (2.5.2). A word that is part of the syntax of a *statement* and that may be used to  
6 identify the *statement*.
- 7 **statement label** (3.2.5). A *lexical token* consisting of up to five digits that precedes a *statement* and  
8 may be used to refer to the *statement*.
- 9 **storage association** (14.6.3). The relationship between two *storage sequences* if a *storage unit* of one  
10 is the same as a *storage unit* of the other.
- 11 **storage sequence** (14.6.3.1). A sequence of contiguous *storage units*.
- 12 **storage unit** (14.6.3.1, 2.4.9). A *character storage unit*, or a *numeric storage unit*, or an *unspecified storage*  
13 *unit*.
- 14 **stride** (6.2.2.3.1). The increment specified in a *subscript triplet*.
- 14 **structure** (2.4.1.2). A *scalar data object* of *derived type*.
- 15 **structure component** (6.1.2). The part of a *structure* corresponding to a *component* of its *type*.
- 16 **subobject** (2.4.3.2). A portion of a *named data object* that may be *referenced* or *defined* independently  
17 of other portions. It may be an *array element*, an *array section*, a *structure component*, or a *substring*.
- 18 **subobject designator** (2.5.1). A *name*, followed by one of more of the following: *component selec-*  
19 *tors*, *array section selectors*, *array element selectors*, and *substring selectors*.
- 20 **subprogram** (2.2). A *function subprogram* or a *subroutine subprogram*. Note that in FORTRAN 77, a  
21 *block data program unit* was called a *subprogram*.
- 22 **subroutine** (2.2.3). A *procedure* that is *invoked* by a *CALL statement* or by a *defined assignment state-*  
23 *ment*.
- 24 **subroutine subprogram** (12.5.2.3). A sequence of *statements* beginning with a *SUBROUTINE* *state-*  
25 *ment* that is not in an *interface block* and ending with the corresponding *END statement*.
- 26 **subscript** (6.2.2). One of the list of *scalar integer expressions* in an *array element selector*. Note that in  
27 FORTRAN 77, the whole list was called the *subscript*.
- 28 **subscript triplet** (6.2.2). An item in the list of an *array section selector* that contains a colon and  
29 specifies a regular sequence of integer values.
- 30 **substring** (6.1.1). A contiguous portion of a *scalar character string*. Note that an *array section* can  
31 include a *substring selector*; the result is called an *array section* and not a *substring*.
- 32 **target** (5.1.2.8). A *named data object* specified in a *type declaration statement* containing the *TARGET*  
33 *attribute*, a *data object* created by an *ALLOCATE statement* for a *pointer*, or a *subobject* of such an  
34 *object*.
- 35 **transformational function**. An *intrinsic function* that is neither an *elemental function* nor an *inquiry*  
36 *function*. It usually has *array arguments* and an *array result* whose elements have values that  
37 depend on the values of many of the elements of the arguments.
- 38 **type** (4). *Data type*.
- 39 **type declaration statement** (5). An *INTEGER*, *REAL*, *DOUBLE PRECISION*, *COMPLEX*, *CHAR-*  
40 *ACTER*, *LOGICAL*, or *TYPE (type-name) statement*.
- 41 **type parameter** (2.4.1.1). A parameter of an *intrinsic data type*. *KIND=* and *LEN=* are the *type*  
42 *parameters*.
- 43 **type parameter values** (4.3). The values of the *type parameters* of a *data entity* of an *intrinsic data*  
44 *type*.

## Glossary of Technical Terms

- 1 **undefined** (2.5.4). For a *data object*, the property of not having a determinate value.
- 2 **use association** (14.6.1.2). The association of *names* in different *scoping units* specified by a *USE*
- 3 *statement*.
- 4 **variable** (2.4.5). A *data object* whose value can be *defined* and redefined during the execution of an
- 5 *executable program*. It may be a *named data object*, an *array element*, an *array section*, a *structure com-*
- 6 *ponent*, or a *substring*. Note that in FORTRAN 77, a variable was always *scalar* and *named*.
- 7 **vector subscript** (6.2.2.3.2). A *section subscript* that is an *integer expression* of rank one.
- 8 **whole array** (6.2.1). A *named array*.

unspecified storage unit (14.6.3.1) A unit of storage for holding  
- pointer or a scalar that is not a pointer and is of type  
other than default integer, default character, default real,  
default double precision, default logical or default complex.

## ANNEX B. DECREMENTAL FEATURES

1 (This Annex is not part of this standard, but is included for information only.)

2 **B.1 Deleted Features.** The deleted features are those features of FORTRAN 77 that are redun-  
3 dant and considered largely unused. Section 1.6.1 describes the nature of the deleted features. The  
4 list of deleted features in this standard is empty.

5 **B.2 Obsolescent Features.** The obsolescent features are those features of FORTRAN 77 that are  
6 redundant and for which better methods are available in FORTRAN 77. Section 1.6.2 describes the  
7 nature of obsolescent features. The obsolescent features are:

- 8 (1) Arithmetic IF — use the IF statement (8.1.2.4) or IF construct (8.1.2)
- 9 (2) Real and double precision DO control variables and DO loop control expressions — use  
10 integer (8.1.4.1)
- 11 (3) Shared DO termination and termination on a statement other than END DO or CON-  
12 TINUE — use an END DO or a CONTINUE statement for each DO statement
- 13 (4) Branching to an END IF statement from outside its IF block — branch to the statement  
14 following the END IF
- 15 (5) Alternate return — see B.2.1
- 16 (6) PAUSE statement — see B.2.2
- 17 (7) ASSIGN and assigned GO TO statements — see B.2.3
- 18 (8) Assigned FORMAT specifiers — see B.2.4
- 19 (9) cH edit descriptor — see B.2.5

20 **B.2.1 Alternate Return.** An alternate return introduces labels into an argument list to allow the  
21 called procedure to direct the execution of the caller upon return. The same effect can be achieved  
22 with a return code that is used in a computed GO TO statement or CASE construct on return. This  
23 avoids an irregularity in the syntax and semantics of argument association. For example,

24 CALL SUBR\_NAME (X, Y, Z, \*100, \*200, \*300)

25 may be replaced by

26 CALL SUBR\_NAME (X, Y, Z, RETURN\_CODE)

27 SELECT CASE (RETURN\_CODE)

28 CASE (1)

29 ...

30 CASE (2)

31 ...

32 CASE (3)

33 ...

34 CASE DEFAULT

35 ...

36 END SELECT

37 **B.2.2 PAUSE Statement.** Execution of a PAUSE statement requires operator or system-specific  
38 intervention to resume execution. In most cases, the same functionality can be achieved as effec-  
39 tively and in a more portable way with the use of an appropriate READ statement that awaits  
40 some input data.

41 **B.2.3 ASSIGN and Assigned GO TO Statements.** The ASSIGN statement allows a label to be  
42 dynamically assigned to an integer variable, and the assigned GO TO statement allows “indirect  
43 branching” through this variable. This hinders the readability of the program flow, especially if  
44 the integer variable also is used in arithmetic operations. The two totally different usages of the

## Decremental Features

- 1 integer variable can be an obscure source of error.
- 2 These statements have commonly been used to simulate internal procedures, which now can be
- 3 coded directly.
  
- 4 **B.2.4 Assigned FORMAT Specifiers.** The ASSIGN statement also allows the label of a FORMAT
- 5 statement to be dynamically assigned to an integer variable, which can later be used as a format
- 6 specifier in READ, WRITE, or PRINT statements. This hinders readability, permits inconsistent
- 7 usage of the integer variable, and can be an obscure source of error.
  
- 8 This functionality is available via character variables, arrays, and constants.
  
- 9 **B.2.5 H editing.** This edit descriptor can be a source of error. The same functionality is available
- 10 using the character constant edit descriptor.



## ANNEX C. SECTION NOTES

1 (This Annex is not part of this standard, but is included for information only.)

### 2 C.1 Section 1 Notes.

3 **C.1.1 Conformance (1.4).** The standard requires a standard-conforming processor to be capable  
4 of detecting and reporting the use within a program unit of forms designated as deleted or obso-  
5 lescent and of additional forms or relationships, where such use can be detected by reference to the  
6 numbered syntax rules and their associated constraints. It is recommended that the processor be  
7 accompanied by documentation that specifies the limits it imposes on the size and complexity of a  
8 program and the means of reporting when these limits are exceeded, that defines the additional  
9 forms and relationships it allows, and that defines the means of reporting the use of additional  
10 forms and relationships and the use of deleted or obsolescent forms. Note that in this context, the  
11 use of a deleted form is the use of an additional form.

12 It is recommended that the processor be accompanied by documentation that specifies the meth-  
13 ods or semantics <sup>of</sup> processor-dependent facilities.

### 14 C.2 Section 2 Notes.

15 **C.2.1 Keywords.** Argument keywords can make procedure references more readable and allow  
16 actual arguments to be in any order. This latter property permits optional arguments. (2.5.2)

### 17 C.3 Section 3 Notes.

18 **C.3.1 Representable Characters (3.1.5).** FORTRAN 77 allowed any character to occur in a charac-  
19 ter context. This standard provides a new feature to allow source programs to contain characters  
20 of more than one kind (4.3.2.1). Characters of different kinds are often identified by control charac-  
21 ters (called "escape" or "shift" characters). It is difficult, if not impossible, for example, to process,  
22 edit, or print files where control characters may not have their intended meaning (as in FORTRAN  
23 77) and where other occurrences may have a control meaning. To provide compatibility with FOR-  
24 TRAN 77 and to allow this standard to meet portability goals, the following approach is incorpo-  
25 rated:

26 (1) In fixed source form, the definition of *rep-char* is not changed.

27 (2) Control characters are not allowed in character contexts in free source form.

28 **C.3.2 Comment Lines (3.3.1.1, 3.3.2.1).** The standard does not restrict the number of consecu-  
29 tive comment lines. The limit on the number of continuation lines permitted for a statement  
30 should not be construed as being a limitation on the number of consecutive comment lines.

31 **C.3.3 Statement Labels (3.2.5).** There are 99999 unique statement labels and a processor must  
32 accept 99999 as a statement label. However, a processor may have an implementation limit on the  
33 total number of unique statement labels in one program unit.

34 **C.3.4 Source Form (3.3).** In fixed source form, an exclamation point (!) in character position 6 is  
35 interpreted as a continuation indicator unless it appears within commentary indicated by a "C" or  
36 "\*" in character position 1 or by another "!" in character positions 1-5. (3.3.2.3)

37 The source form of FORTRAN 77, FORTRAN 66, and the initial Fortran in 1954 was predicated on a  
38 common form of input, the 80-column card. However, on the IBM 704, only 72 columns could be  
39 used and the remaining eight columns were designated as commentary. In some implementations  
40 of FORTRAN 77, these columns are so used. They contain "line numbers" and are used by an editor  
41 to manage changes to a program. (3.3.2)

42 The Fortran Standards Subcommittee believes that 66 positions are inadequate to represent reada-  
43 ble Fortran source code, particularly with "long" names and the use of indentation. Consequently,

## Section Notes

- 1 in the new source form, this standard relaxes the FORTRAN 77 restriction on source line size.
- 2 Given the need for an incompatible new source form in Fortran, additional restrictions of the rigid
- 3 card form are relaxed. Positions six and seven are no longer "special" and the continuation mark
- 4 is on the line being continued rather than on the continuation line. Blank characters are generally
- 5 significant in the new source form, but other features of the new form apply to either form, and are
- 6 allowed in either. (3.3.1)
- 7 The rule allowing optional blanks at specific places in some keywords (for example, ENDIF or
- 8 END IF) is intended to permit a reasonable choice to users accustomed to insignificant blanks.
- 9 In some circumstances, for example where source code is maintained in an INCLUDE file for use
- 10 in programs whose source form might be of either form, observing the following rules allows the
- 11 code to be used with either:
  - 12 (1) Confine statement labels to character positions 1 to 5 and statements to character posi-
  - 13 tions 7 to 72;
  - 14 (2) Treat blanks as being significant;
  - 15 (3) Use only the exclamation mark (!) to indicate a comment, but do not start the comment
  - 16 in column 6;
  - 17 (4) For continued statements, place an ampersand (&) in both character position 73 of a
  - 18 continued line and character position 6 of a continuing line.

### 19 C.4 Section 4 Notes.

20 **C.4.1 Zero (4.3.1).** A processor must not consider a negative zero to be different from a positive  
21 zero.

22 **C.4.2 Characters (4.2).** Free source form allows only graphic characters as representable charac-  
23 ters. Almost all control characters have uses or effects that effectively preclude their use in charac-  
24 ter literals. Nevertheless, for compatibility with FORTRAN 77, control characters remain permitted  
25 in principle in fixed source form.

26 **C.4.3 Intrinsic and Derived Data Types (4.3, 4.4).** FORTRAN 77 provided only data types explic-  
27 itly defined in the standard (logical, integer, real, double precision, complex, and character). This  
28 standard provides those intrinsic types and provides derived types to allow the creation of new  
29 data types. A derived-type definition specifies a data structure consisting of components of intrin-  
30 sic types and of derived types. Such a type definition does not represent a data object, but rather, a  
31 template for declaring named objects of that derived type. For example, the definition

```
32 TYPE POINT  
33     INTEGER X_COORD  
34     INTEGER Y_COORD  
35 END TYPE POINT
```

36 specifies a new derived type named POINT which is composed of two components of intrinsic  
37 type integer (X\_COORD and Y\_COORD). The statement TYPE (POINT) FIRST, LAST declares  
38 two data objects, FIRST and LAST, that can hold values of type POINT.

39 FORTRAN 77 provided REAL and DOUBLE PRECISION intrinsic types as approximations to mathe-  
40 matical real numbers. This standard generalizes REAL as an intrinsic type with a type parameter  
41 that selects the approximation method. The type parameter is named KIND and has values that  
42 are processor dependent. DOUBLE PRECISION is treated as a synonym for REAL (*k*), where *k* is  
43 the implementation-defined kind type parameter value KIND (0.0D0).

44 Real literal constants may be specified with a kind type parameter to ensure that they have a par-  
45 ticular kind type parameter value (4.3.1.2).

1 For example, with the specifications

```
2 INTEGER Q
3 PARAMETER (Q = 8)
4 REAL (Q) B
```

5 the literal constant 10.93\_Q has the same precision as the variable B.

6 X3.9-1978 did not allow zero-length character strings. They are permitted by this standard  
7 (4.3.2.1).

8 Objects are of different derived type if they are declared using different derived-type definitions.  
9 For example,

```
10 TYPE APPLES
11     INTEGER NUMBER
12 END TYPE APPLES
13 TYPE ORANGES
14     INTEGER NUMBER
15 END TYPE ORANGES
16 TYPE (APPLES) COUNT1
17 TYPE (ORANGES) COUNT2
18 COUNT1 = COUNT2 ! Erroneous statement mixing apples and oranges
```

19 Even though all components of objects of type APPLES and objects of type ORANGES have identi-  
20 cal intrinsic types, the objects are of different types.

21 **C.4.4 Selection of the Approximation Methods.** This standard permits the selection of the real  
22 approximation method for an entire program to be parameterized through the use of the parame-  
23 terized real data type and module. This is accomplished by defining a named integer constant, say  
24 FLOAT, to have a specific kind type parameter value, and to use that named constant in all real,  
25 complex, and derived-type declarations. For example, the specification statements

```
26 INTEGER FLOAT
27 PARAMETER (FLOAT = 8)
28 REAL (FLOAT) X, Y
29 COMPLEX (FLOAT) Z
```

30 specify that the approximation method corresponding to a kind type parameter value of 8 is sup-  
31 plied for the data objects X, Y, and Z in the program unit. The kind type parameter value FLOAT  
32 can be made available to an entire program by placing the INTEGER and PARAMETER specifica-  
33 tion statements in a module and accessing the named constant FLOAT with a USE statement.  
34 Note that by changing 8 to 4 once in the module, a different approximation method is selected.

35 To avoid the use of the processor-dependent values 4 or 8, replace 8 by KIND (0.0) or KIND  
36 (0.0D0). Another way to avoid these processor-dependent values is to select the kind value using  
37 the intrinsic inquiry function SELECTED\_REAL\_KIND. This function, given integer arguments P  
38 and R specifying minimum requirements for decimal precision and decimal exponent range,  
39 respectively, returns the kind type parameter value of the approximation method that has at least  
40 P decimal digits of precision and at least a range for positive numbers of  $10^{-R}$  to  $10^R$ . In the above  
41 specification statement, the 8 may be replaced by, for instance, SELECTED\_REAL\_KIND (10, 50),  
42 which requires an approximation method to be selected with at least 10 decimal digits of precision  
43 and an exponent range from  $10^{-50}$  to  $10^{50}$ .

44 There are no magnitude or ordering constraints placed on kind values, in order that implementers  
45 may have flexibility in assigning such values and may add new kinds without changing previ-  
46 ously assigned kind values.

47 As kind values have no portable meaning, a good practice is to use them in programs only through  
48 named constants as described above (for example, SINGLE, IEEE\_SINGLE, DOUBLE, and QUAD),  
49 rather than using the kind values directly.

## Section Notes

1 **C.4.5 Storage of Derived Types (4.4.1).** A structure resolves into a sequence of components of  
2 intrinsic type. Unless the structure includes a SEQUENCE statement, the use of this terminology  
3 in no way implies that these components are stored in this, or any other, order. Nor is there any  
4 requirement that contiguous storage be used. The sequence merely refers to the fact that in writ-  
5 ing the definitions there will necessarily be an order in which the components appear, and this will  
6 define a sequence of components. This order is of limited significance since a component of an  
7 object of derived type will always be accessed by a component name except in the following con-  
8 texts: the sequence of expressions in a derived-type value constructor, the data values in namelist  
9 input data, and the inclusion of the structure in an input/output list of a formatted data transfer,  
10 where it is expanded to this sequence of components. Provided the processor adheres to the  
11 defined order in these cases, it is otherwise free to organize the storage of the components for any  
12 structure in memory as best suited to the particular architecture.

13 **C.4.6 Pointers.** This standard introduces pointers as names that can change dynamically their  
14 association with a target object. In a sense, a normal variable is a name with a fixed association  
15 with a specific object. A normal variable name refers to the same storage space throughout the  
16 lifetime of a variable. A pointer name may refer to different storage space, or even no storage  
17 space, at different times. A variable may be considered to be a descriptor for space to hold values  
18 of the appropriate type, type parameters, and array rank such that the values stored in the descrip-  
19 tor are fixed when the variable is created by its declaration. A pointer also may be considered to  
20 be a descriptor, but one whose values may be changed dynamically so as to describe different  
21 pieces of storage. When a pointer is declared, space to hold the descriptor is created, but the space  
22 for the target object is not created.

23 A derived type may have one or more components that are defined to be pointers. It may have a  
24 component that is a pointer to an object of the same derived type. This "recursive" data definition  
25 allows dynamic data structures such as linked lists, trees, and graphs to be constructed. For exam-  
26 ple,

```
27 TYPE CELL          ! Define a "recursive" type
28     INTEGER :: VAL
29     TYPE (CELL), POINTER :: NEXT_CELL
30 END TYPE CELL

31 TYPE (CELL), TARGET :: HEAD
32 TYPE (CELL), POINTER :: CURRENT, TEMP ! Declare pointers
33 INTEGER :: IOEM, K

34 HEAD % VAL = 0
35 CURRENT => HEAD ! CURRENT points to head of list
36 DO NULLIFY (CURRENT % NEXT_CELL)
37     READ (*, *, IOSTAT = IOEM) K ! Read next value, if any
38     IF (IOEM /= 0) EXIT
39     ALLOCATE (TEMP) ! Create new cell each iteration
40     TEMP % VAL = K ! Assign value to cell
41     NULLIFY (TEMP % NEXT_CELL) ! Set status to disassociated
42     CURRENT % NEXT_CELL => TEMP ! Attach new cell to list
43     CURRENT => TEMP ! CURRENT points to new end of list
44 END DO
```

45 A list is now constructed and the last linked cell contains a disassociated pointer. A loop can be  
46 used to "walk through" the list.

```

1 CURRENT => HEAD
2 DO
3   WRITE (*, *) CURRENT % VAL
4   IF (.NOT. ASSOCIATED (CURRENT % NEXT_CELL)) EXIT
5   CURRENT => CURRENT % NEXT_CELL
6 END DO

```

## 7 C.5 Section 5 Notes.

8 **C.5.1 Type Declaration Statements (5.1).** Type declaration statements in FORTRAN 77 required  
9 different attributes of an entity to be specified in different statements (INTEGER, SAVE, DATA,  
10 etc.). This standard allows the attributes of an entity to be specified in a single extended form of  
11 the type statement. For example,

```

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

```

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

16 **C.5.2 The POINTER Attribute (5.1.2.7).** The pointer attribute must be specified to declare a  
17 pointer. The type, type parameters, and rank, which may be specified in the same statement or  
18 with one or more attribute specification statements, determine the characteristics of the target  
19 objects that may be associated with the pointers declared in the statement. An obvious model for  
20 interpreting declarations of pointers is that such declarations create for each name a descriptor.  
21 Such a descriptor includes all the data necessary to describe fully and locate in memory an object  
22 and all subobjects of the type, type parameters, and rank specified. The descriptor is created  
23 empty; it does not contain values describing how to access an actual memory space. These  
24 descriptor values will be filled in when the pointer is associated with actual target space.

25 The following example illustrates the use of pointers in an iterative algorithm:

```

26 PROGRAM DYNAM_ITER
27   REAL, DIMENSION (:, :), POINTER :: A, B, SWAP ! Declare pointers
28   ...
29   READ (*, *) N, M
30   ALLOCATE (A (N, M), B (N, M)) ! Allocate target arrays
31   ! Read values into A
32   ...
33   ITER: DO
34     ...
35     ! Apply transformation of values in A to produce values in B
36     ...
37     IF (CONVERGED) EXIT ITER
38     ! Swap A and B
39     SWAP => A; A => B; B => SWAP
40   END DO ITER
41   ...
42 END

```

43 **C.5.3 The TARGET Attribute (5.1.2.8).** The TARGET attribute must be specified for any non-  
44 pointer object that may, during the execution of the program, become associated with a pointer.  
45 This attribute is defined solely for optimization purposes. It allows the processor to assume that  
46 any nonpointer object not explicitly declared as a target may be referred to only by way of its origi-  
47 nal declared name. In particular, it means that implicitly-declared objects must not be used as  
48 pointer targets. This will allow a processor to perform optimizations that otherwise would not be  
49 possible in the presence of certain pointers.

## Section Notes

1 The following example illustrates the use of the TARGET attribute in an iterative algorithm:

```
2 PROGRAM ITER
3   REAL, DIMENSION (1000, 1000), TARGET :: A, B
4   REAL, DIMENSION (:, :), POINTER      :: IN, OUT, SWAP
5   ...
6   ! Read values into A
7   ...
8   IN => A           ! Associate IN with target A
9   OUT => B          ! Associate OUT with target B
10  ...
11  ITER:DO
12    ...
13    ! Apply transformation of IN values to produce OUT
14    ...
15    IF (CONVERGED) EXIT ITER
16    ! Swap IN and OUT
17    SWAP => IN; IN => OUT; OUT => SWAP
18  END DO ITER
19  ...
20  END
```

21 **C.5.4 PARAMETER Statements and IMPLICIT NONE (5.2.10, 5.3).** Because an implicitly typed  
22 named constant may precede an IMPLICIT statement only if that IMPLICIT statement serves to  
23 confirm the type of the named constant, it follows that if an IMPLICIT NONE statement is to  
24 appear, it must precede all PARAMETER statements.

25 **C.5.5 EQUIVALENCE Statement Extensions (5.5.1).** The EQUIVALENCE statement has been  
26 extended to allow the equivalencing of sequence structures and the equivalencing of objects of  
27 intrinsic type with nondefault type parameters, but there are strict rules regarding the appearance  
28 of these objects in an EQUIVALENCE statement.

29 Structures that appear in EQUIVALENCE statements must be sequence structures. If a sequence  
30 structure is not of numeric sequence type or of character sequence type, it must be equivalenced  
31 only to objects of the same type.

32 A numeric sequence structure may be equivalenced to another numeric sequence structure, an  
33 object of default integer type, default real type, double precision real type, default complex type,  
34 or default logical type such that components of the structure ultimately become associated only  
35 with objects of these types.

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

38 Other objects may be equivalenced only to objects of the same type and kind type parameters.

39 **C.5.6 COMMON Statement Extensions (5.5.2).** Modules provide global access to all objects;  
40 however, the COMMON statement also has been extended to allow access in more than one scop-  
41 ing unit to objects with the POINTER attribute, to sequence structures, and to objects of intrinsic  
42 type with nondefault type parameters.

43 A common block is permitted to contain sequences of different storage units, provided each scop-  
44 ing unit that accesses the common block specifies an identical sequence of storage units for the  
45 common block. This extension allows a single common block to contain both numeric and charac-  
46 ter objects.

47 Association in different scoping units between objects of default type, objects of double precision  
48 real type, and sequence structures is permitted according to the rules for equivalence objects  
49 (5.5.1).

1 **C.6 Section 6 Notes.**

2 **C.6.1 Substrings (6.1.1).** Substrings are of zero length when the starting point exceeds the end-  
 3 ing point. This was not allowed in FORTRAN 77. This standard also allows substrings of literal char-  
 4 acter constants and named character constants.

5 **C.6.2 Array Element References (6.2.2).** A subscript reference to an element outside the  
 6 declared bounds is not standard conforming, as in FORTRAN 77.

7 **C.6.3 Structure Components (6.1.2).** Components of a structure are referenced by writing the  
 8 components of successive levels of the structure hierarchy until the desired component is  
 9 described. For example,

```
10 TYPE ID_NUMBERS
11     INTEGER SSN
12     INTEGER EMPLOYEE_NUMBER
13 END TYPE ID_NUMBERS
```

```
14 TYPE PERSON_ID
15     CHARACTER (LEN=30) LAST_NAME
16     CHARACTER (LEN=1) MIDDLE_INITIAL
17     CHARACTER (LEN=30) FIRST_NAME
18     TYPE (ID_NUMBERS) NUMBER
19 END TYPE PERSON_ID
```

```
20 TYPE PERSON
21     INTEGER AGE
22     TYPE (PERSON_ID) ID
23 END TYPE PERSON
```

```
24 TYPE (PERSON) GEORGE, MARY
```

```
25 PRINT *, GEORGE % AGE           ! Print the AGE component
26 PRINT *, MARY % ID % LAST_NAME  ! Print LAST_NAME of MARY
27 PRINT *, MARY % ID % NUMBER % SSN ! Print SSN of MARY
28 PRINT *, GEORGE % ID % NUMBER  ! Print SSN and EMPLOYEE_NUMBER of GEORGE
```

29 A structure component may be a data object of intrinsic type as in the case of GEORGE % AGE or  
 30 it may be of derived type as in the case of GEORGE % ID % NUMBER. The resultant component  
 31 may be a scalar or an array of intrinsic or derived type.

```
32 TYPE LARGE
33     INTEGER ELT (10)
34     INTEGER VAL
35 END TYPE LARGE
```

```
36 TYPE (LARGE) A (5)           ! 5 element array, each of whose elements
37                               ! includes a 10 element array ELT and
38                               ! a scalar VAL.
39 PRINT *, A (1)               ! Prints 10 element array ELT and scalar VAL.
40 PRINT *, A (1) % ELT (3)    ! Prints scalar element 3
41                               ! of array element 1 of A.
42 PRINT *, A (2:4) % VAL      ! Prints scalar VAL for array elements
43                               ! 2 to 4 of A.
```

## Section Notes

1 **C.6.4 Pointer Allocation and Association.** The effect of ALLOCATE, DEALLOCATE, NULLIFY, and pointer assignment is that they are interpreted as changing the values in the descriptor  
2 that is the pointer. An ALLOCATE is assumed to create space for a suitable object and to "assign"  
3 to the pointer the values necessary to describe that space. A NULLIFY breaks the association of  
4 the pointer with the space. A DEALLOCATE breaks the association and releases the space.  
5 Depending on the implementation, it could be seen as setting a flag in the pointer that indicates  
6 whether the values in the descriptor are valid, or it could clear the descriptor values to some (say  
7 zero) value indicative of the pointer not currently pointing to anything. A pointer assignment cop-  
8 ies the values necessary to describe the space occupied by the target into the descriptor that is the  
9 pointer. Descriptors are copied, values of objects are not.

11 If PA and PB are both pointers and PB currently is associated with a target, then

12 PA => PB

13 results in PA being associated with the same target as PB. If PB was disassociated, then PA  
14 becomes disassociated.

15 The standard is specified so that such associations are direct and independent. A subsequent state-  
16 ment

17 PB => D

18 or

19 ALLOCATE (PB)

20 has no effect on the association of PA with its target. A statement

21 DEALLOCATE (PB)

22 leaves PA as a "dangling pointer" to space that has been released. The program must not use PA  
23 again until it becomes associated via pointer assignment or an ALLOCATE statement.

24 DEALLOCATE should only be used to release space that was created by a previous ALLOCATE.  
25 Thus the following is invalid:

26 REAL, TARGET :: T

27 REAL, POINTER :: P

28 ...

29 P => T

30 DEALLOCATE (P) ! Not allowed: P's target was not allocated

31 The basic principle is that ALLOCATE, NULLIFY, and pointer assignment primarily affect the  
32 pointer rather than the target. ALLOCATE creates a new target but, other than breaking its con-  
33 nection with the specified pointer, it has no effect on the old target. Neither NULLIFY nor pointer  
34 assignment has any effect on targets. A given piece of memory that was allocated and associated  
35 with a pointer will become inaccessible to a program if the pointer is nullified and no other pointer  
36 was associated with this piece of memory. Such pieces of memory may be reused by the processor  
37 if this is expedient. However, whether such inaccessible memory is in fact reused is entirely proc-  
38 essor dependent.



## 1 C.6.5 Partial Summary of Array Name Appearances.

2 Table C.1 Some of the Possible Appearances of Array Names

Priority:  
1) complete  
2) incomplete  
3) gone  
4) wrong.

| Place of Appearance                               | Explicit Shape Array | Structure Component Array | Target Array | Allocatable Array | Assumed Shape Array | Assumed Size Array |
|---------------------------------------------------|----------------------|---------------------------|--------------|-------------------|---------------------|--------------------|
| <i>dummy-arg</i>                                  | Yes                  | No                        | Yes          | No                | Yes                 | Yes                |
| <i>use-stmt</i>                                   | Yes                  | No                        | Yes          | Yes               | No                  | No                 |
| <i>save-stmt</i>                                  | Yes                  | No                        | Yes          | Yes               | No                  | No                 |
| <i>namelist-stmt</i>                              | Yes                  | No                        | No           | No                | No                  | No                 |
| <i>equivalence-stmt</i>                           | Yes                  | No                        | No           | No                | No                  | No                 |
| <i>data-stmt</i>                                  | Yes                  | No                        | No           | No                | No                  | No                 |
| <i>common-stmt</i>                                | Yes                  | No                        | No           | No                | No                  | No                 |
| <i>type-declaration-stmt</i>                      | Yes                  | Yes                       | Yes          | Yes               | Yes                 | Yes                |
| <i>format</i>                                     | Yes                  | Yes                       | Yes          | Yes               | Yes                 | Yes                |
| <i>actual-arg</i> in a reference to a procedure   | Yes                  | Yes                       | Yes          | Yes               | Yes                 | Yes                |
| <i>input-item-list</i> or <i>output-item-list</i> | Yes                  | Yes                       | Yes          | Yes               | Yes                 | No                 |
| <i>internal-file-unit</i>                         | Yes                  | Yes                       | Yes          | Yes               | Yes                 | No                 |
| <i>primary</i>                                    | Yes                  | Yes                       | Yes          | Yes               | Yes                 | No                 |
| <i>assignment-stmt</i>                            | Yes                  | Yes                       | Yes          | Yes               | Yes                 | No                 |
| <i>vector-subscript</i>                           | Yes                  | Yes                       | Yes          | Yes               | Yes                 | No                 |
| <i>pointer-assignment-stmt</i>                    | No                   | No                        | Yes          | No                | No                  | No                 |
| <i>allocate-stmt</i>                              | No                   | No                        | Yes          | Yes               | No                  | No                 |
| <i>deallocate-stmt</i> or <i>nullify-stm</i>      | No                   | No                        | Yes          | Yes               | No                  | No                 |
| <i>function name, result name, or entry name</i>  | Yes                  | No                        | Yes          | No                | No                  | No                 |

## 35 C.7 Section 7 Notes.

36 C.7.1 Character Assignment. The FORTRAN 77 restriction that none of the character positions  
37 being defined in the character assignment statement may be referenced in the expression has been  
38 removed (7.5.1.5).

39 C.7.2 Evaluation of Function References. If more than one function reference appears in a  
40 statement, they may be executed in any order (subject to a function result being evaluated after the  
41 evaluation of its arguments) and their values must not depend on the order of execution. This lack  
42 of dependence on order of evaluation permits parallel execution of the function references (7.1.7.1).

43 C.7.3 Pointers in Expressions. A pointer is basically considered to be like any other variable  
44 when it is used as a primary in an expression. If a pointer is used as an operand to an operator  
45 that expects a value, the pointer will automatically deliver the value contained in the space cur-  
46 rently described by the pointer, that is, the value of the target object currently associated with the  
47 pointer. In value-demanding expression contexts, pointers are dereferenced.

## Section Notes

1 **C.7.4 Pointers on the Left Side of an Assignment.** A pointer that appears on the left of an  
2 intrinsic assignment statement also is dereferenced and is taken to be referring to the space that is  
3 its current target. Therefore, the assignment statement specifies the normal copying of the value of  
4 the right-hand expression into this target space. All the normal rules of intrinsic assignment hold;  
5 the type and type parameters of the expression and the pointer target must agree and the shapes  
6 must be conformable.

7 For intrinsic assignment of derived types, nonpointer components are assigned and pointer com-  
8 ponents are pointer assigned. Dereferencing is applied only to entire scalar objects, not selectively  
9 to pointer subobjects.

10 For example, suppose a type such as

```
11 TYPE CELL  
12     INTEGER :: VAL  
13     TYPE (CELL), POINTER :: NEXT_CELL  
14 ENDTYPE
```

15 is defined and objects such as HEAD and CURRENT are declared using

```
16 TYPE (CELL), TARGET :: HEAD  
17 TYPE (CELL), POINTER :: CURRENT
```

18 If a linked list has been created and attached to HEAD and the pointer CURRENT has been allo-  
19 cated space, statements such as

```
20 CURRENT = HEAD  
21 CURRENT = CURRENT % NEXT_CELL
```

22 cause the contents of the cells referenced on the right to be copied to the cell referred to by CUR-  
23 RENT. In particular, the right-hand side of the second statement causes the pointer component in  
24 the cell, CURRENT, to be selected. This pointer is dereferenced because it is in an expression con-  
25 text to produce the target's integer value and a pointer to a cell that is contained in the target's  
26 NEXT\_CELL component. The left-hand side causes the pointer CURRENT to be dereferenced to  
27 produce its present target, namely space to hold a cell (an integer and a cell pointer). The integer  
28 value on the right is copied to the integer space on the left and the pointer components are pointer  
29 assigned (the descriptor on the right is copied into the space for a descriptor on the left). When a  
30 statement such as

```
31 CURRENT => CURRENT % NEXT_CELL
```

32 is executed, the descriptor value in CURRENT % NEXT\_CELL is copied to the descriptor named  
33 CURRENT. In this case, CURRENT is made to point at a different target.

34 In the intrinsic assignment statement, the space associated with the current pointer does not  
35 change but the values stored in that space do. In the pointer assignment, the current pointer is  
36 made to associate with different space. Using the intrinsic assignment causes a linked list of cells  
37 to be moved up through the current "window"; the pointer assignment causes the current pointer  
38 to be moved down through the list of cells.

## 39 C.8 Section 8 Notes.

40 **C.8.1 Loop Control.** Fortran provides several forms of loop control:

- 41 (1) With an iteration count and a DO variable. This is the classic Fortran DO loop.
- 42 (2) Test a logical condition before each execution of the loop (DO WHILE).
- 43 (3) DO "forever".

1 **C.8.2 The CASE Construct.** At most one case block is selected for execution within a CASE construct, and there is no fall-through from one block into another block within a CASE construct.  
 2  
 3 Thus there is no requirement for the user to exit explicitly from a block.

4 **C.8.3 Examples of Invalid DO Constructs.** The following are all examples of invalid skeleton  
 5 DO constructs:

6 Example 1:

```
7 DO I = 1, 10
8   ...
9 END DO LOOP ! No matching construct name
```

10 Example 2:

```
11 LOOP: DO 1000 I = 1, 10 ! No matching construct name
12   ...
13 1000 CONTINUE
```

14 Example 3:

```
15 LOOP1: DO
16   ...
17 END DO LOOP2 ! Construct names don't match
```

18 Example 4:

```
19 DO I = 1, 10 ! Label required or ...
20   ...
21 1010 CONTINUE ! ... END DO required
```

22 Example 5:

```
23 DO 1020 I = 1, 10
24   ...
25 1021 END DO ! Labels don't match
```

26 Example 6:

```
27 FIRST: DO I = 1, 10
28   SECOND: DO J = 1, 5
29   ...
30   END DO FIRST ! Improperly nested DOs
31 END DO SECOND
```

## 32 C.9 Section 9 Notes.

33 **C.9.1 Input/Output Records (9.1).** What is called a "record" in Fortran is commonly called a  
 34 "logical record". There is no concept in Fortran of a "physical record".

35 An endfile record does not necessarily have any physical embodiment. The processor may use a  
 36 record count or other means to register the position of the file at the time an ENDFILE statement is  
 37 executed, so that it can take appropriate action when that position is reached again during a read  
 38 operation. The endfile record, however it is implemented, is considered to exist for the BACK-  
 39 SPACE statement (9.1.3).

40 **C.9.2 Files (9.2).** This standard accommodates, but does not require, file cataloging. To do this,  
 41 several concepts are introduced.

42 **C.9.2.1 File Connection (9.3).** Before any input/output may be performed on a file, it must be  
 43 connected to a unit. The unit then serves as a designator for that file as long as it is connected. To  
 44 be connected does not imply that "buffers" have or have not been allocated, that "file-control  
 45 tables" have or have not been filled out, or that any other method of implementation has been

## Section Notes

1 used. Connection means that (barring some other fault) a READ or WRITE statement may be exe-  
2 cuted on the unit, hence on the file. Without a connection, a READ or WRITE statement must not  
3 be executed.

4 **C.9.2.2 File Existence (9.2.1.1).** Totally independent of the connection state is the property of  
5 existence, this being a file property. The processor "knows" of a set of files that exist at a given  
6 time for a given executable program. This set would include tapes ready to read, files in a catalog,  
7 a keyboard, a printer, etc. The set may exclude files inaccessible to the executable program  
8 because of security, because they are already in use by another executable program, etc. This  
9 standard does not specify which files exist, hence wide latitude is available to a processor to imple-  
10 ment security, locks, privilege techniques, etc. Existence is a convenient concept to designate all of  
11 the files that an executable program can potentially process.

12 All four combinations of connection and existence may occur:

| Connect | Exist | Examples                                             |
|---------|-------|------------------------------------------------------|
| Yes     | Yes   | A card reader loaded and ready to be read            |
| Yes     | No    | A printer before the first line is written           |
| No      | Yes   | A file named 'JOAN' in the catalog                   |
| No      | No    | A file on a reel of tape, not known to the processor |

23 Means are provided to create, delete, connect, and disconnect files.

24 **C.9.2.3 File Names (9.3.4.1).** A file may have a name. The form of a file name is not specified.  
25 If a system does not have some form of cataloging or tape labeling for at least some of its files, all  
26 file names will disappear at the termination of execution. This is a valid implementation.  
27 Nowhere does this standard require names to survive for any period of time longer than the exe-  
28 cution time span of an executable program. Therefore, this standard does not impose cataloging as  
29 a prerequisite. The naming feature is intended to allow use of a cataloging system where one  
30 exists.

31 **C.9.2.4 File Access (9.2.1.2).** This standard does not address problems of security, protection,  
32 locking, and many other concepts that may be part of the concept of "right of access". Such con-  
33 cepts are considered to be in the province of an operating system.

34 The OPEN and INQUIRE statements can be extended naturally to consider these things.

35 Possible access methods for a file are: sequential and direct. The processor may implement two  
36 different types of files, each with its own access method. It might also implement one type of file  
37 with two different access methods.

38 Direct access to files is of a simple and commonly available type, that is, fixed-length records. The  
39 key is a positive integer.

40 **C.9.2.5 Nonadvancing Input/Output (9.2.1.3.1).** Data transfer statements affect the positioning  
41 of an external file. In FORTRAN 77, if no error or end-of-file condition exists, the file is positioned  
42 after the record just read or written and that record becomes the preceding record. This standard  
43 contains the record positioning ADVANCE= specifier in a data transfer statement that provides  
44 the capability of maintaining a position within the current record from one formatted data transfer  
45 statement to the next data transfer statement. The value NO provides this capability. The value  
46 YES positions the file after the record just read or written. The default is YES.

47 The tab edit descriptor and the slash are still appropriate for use with this type of record access but  
48 the tab will not reposition before the left tab limit.

- 1 A BACKSPACE of a file that is currently positioned within a record causes the specified unit to be  
2 positioned before the current record.
- 3 If the last data transfer statement was WRITE and the file is currently positioned within a record,  
4 the file will be positioned implicitly after the current record before an ENDFILE record is written  
5 to the file, that is, a REWIND, BACKSPACE, or ENDFILE statement following a nonadvancing  
6 WRITE statement causes the file to be positioned at the end of the current output record before the  
7 endfile record is written to the file.
- 8 This standard provides a SIZE= specifier to be used with nonadvancing data transfer statements.  
9 The variable in the SIZE= specifier will contain the count of the number of characters that make up  
10 the sequence of values read by the data edit descriptors in this input statement.
- 11 The count is especially helpful if there is only one list item in the input list since it will contain the  
12 number of characters that were present for the item.
- 13 The EOR= specifier is provided to indicate when an end-of-record condition has been encountered  
14 during a nonadvancing data transfer statement. The end-of-record condition is not an error condi-  
15 tion. If this specifier is present, the current input list item that required more characters than the  
16 record contained will be padded with blanks if PAD= 'YES' is in effect. This means that the iolist  
17 item was successfully completed. The file will then be positioned after the current record. The  
18 IOSTAT= specifier, if present, will be defined with a processor-dependent negative value and the  
19 data transfer statement will be terminated. Program execution will continue with the statement  
20 specified in the EOR= specifier. The EOR= specifier gives the capability of taking control of execu-  
21 tion when the end-of-record has been found. Implied-DO variables retain their last defined value  
22 and any remaining items in the iolist retain their definition status when an end-of-record condition  
23 occurs. The SIZE= specifier, if present, will contain the number of characters read with the data  
24 edit descriptors during this READ statement.
- 25 For nonadvancing input, the processor is not required to read partial records. The processor may  
26 read the entire record into an internal buffer and make successive portions of the record available  
27 to successive input statements.
- 28 **C.9.3 OPEN Statement (9.3.4).** A file may become connected to a unit in either of two ways: pre-  
29 connection or execution of an OPEN statement. Preconnection is performed prior to the beginning  
30 of execution of an executable program by means external to Fortran. For example, it may be done  
31 by job control action or by processor-established defaults. Execution of an OPEN statement is not  
32 required to access preconnected files (9.3.3).
- 33 The OPEN statement provides a means to access existing files that are not preconnected. An  
34 OPEN statement may be used in either of two ways: with a file name (open-by-name) and without  
35 a file name (open-by-unit). A unit is given in either case. Open-by-name connects the specified file  
36 to the specified unit. Open-by-unit connects a processor-determined default file to the specified  
37 unit. (The default file may or may not have a name.)
- 38 Therefore, there are three ways a file may become connected and hence processed: preconnection,  
39 open-by-name, and open-by-unit. Once a file is connected, there is no means in standard Fortran  
40 to determine how it became connected.
- 41 An OPEN statement may also be used to create a new file. In fact, any of the foregoing three con-  
42 nection methods may be performed on a file that does not exist. When a unit is preconnected,  
43 writing the first record creates the file. With the other two methods, execution of the OPEN state-  
44 ment creates the file.
- 45 When an OPEN statement is executed, the unit specified in the OPEN may or may not already be  
46 connected to a file. If it is already connected to a file (either through preconnection or by a prior  
47 OPEN), then omitting the FILE= specifier in the OPEN statement implies that the file is to remain  
48 connected to the unit. Such an OPEN statement may be used to change the values of the BLANK=,  
49 DELIM=, or PAD= specifiers.
- 50 Note that, since an OPEN that specifies STATUS = 'SCRATCH' is not allowed to have a FILE=  
51 specifier, such an OPEN always attempts to retain any connection that the specified unit may have.

## Section Notes

1 If the unit were already connected to a file, and if that connection did not have a STATUS of  
2 SCRATCH, then the OPEN would be illegal because the value of the STATUS= specifier must not  
3 be changed by the OPEN.

4 The following examples illustrate these rules. In the first example, unit 10 is preconnected to a  
5 SCRATCH file; the OPEN statement changes the value of PAD= to YES.

```
6 CHARACTER (LEN = 20) CH1
7 WRITE (10, '(A)') 'THIS IS RECORD 1'
8 OPEN (UNIT = 10, STATUS = 'SCRATCH', PAD = 'YES')
9 REWIND 10
10 READ (10, '(A20)') CH1 ! CH1 now has the value
11 ! 'THIS IS RECORD 1'
```

12 In the next example, unit 12 is first connected to a file named FRED, with a status of OLD. The sec-  
13 ond OPEN statement then opens unit 12 again, retaining the connection to the file FRED, but  
14 changing the value of the DELIM= specifier to QUOTE.

```
15 CHARACTER (LEN = 25) CH2, CH3
16 OPEN (12, FILE = 'FRED', STATUS = 'OLD', DELIM = 'NONE')
17 CH2 = '"THIS STRING HAS QUOTES."'
18 ! Quotes in string CH2
19 WRITE (12, *) CH2 ! Written with no delimiters
20 OPEN (12, DELIM = 'QUOTE') ! Now quote is the delimiter
21 REWIND 12
22 READ (12, *) CH3 ! CH3 now has the value
23 ! 'THIS STRING HAS QUOTES.'
```

24 The next example is invalid because it attempts to change the value of the STATUS= specifier.

```
25 OPEN (10, FILE = 'FRED', STATUS = 'OLD')
26 WRITE (10, *) A, B, C
27 OPEN (10, STATUS = 'SCRATCH') ! Attempts to make FRED
28 ! a SCRATCH file
```

29 The previous example could be made valid by closing the unit first, as in the next example.

```
30 OPEN (10, FILE = 'FRED', STATUS = 'OLD')
31 WRITE (10, *) A, B, C
32 CLOSE (10)
33 OPEN (10, STATUS = 'SCRATCH') ! Opens a different
34 ! SCRATCH file
```

35 **C.9.4 Connection Properties (9.3.2).** When a unit becomes connected to a file, either by execu-  
36 tion of an OPEN statement or by preconnection, the following connection properties may be estab-  
37 lished:

- 38 (1) An access method, which is sequential or direct, is established for the connection  
39 (9.3.4.3).
- 40 (2) A form, which is formatted or unformatted, is established for a connection to a file that  
41 exists or is created by the connection. For a connection that results from execution of an  
42 OPEN statement, a default form (which depends on the access method, as described in  
43 9.2.1.2) is established if no form is specified. For a preconnected file that exists, a form is  
44 established by preconnection. For a preconnected file that does not exist, a form may be  
45 established, or the establishment of a form may be delayed until the file is created (for  
46 example, by execution of a formatted or unformatted WRITE statement) (9.3.4.4).
- 47 (3) A record length may be established. If the access method is direct, the connection estab-  
48 lishes a record length that specifies the length of each record of the file. An existing file  
49 with records that are not all of equal length must not be connected for direct access.
- 50 If the access method is sequential, records of varying lengths are permitted. In this case,

- 1 the record length established specifies the maximum length of a record in the file  
2 (9.3.4.5).
- 3 (4) A blank significance property, which is ZERO or NULL, is established for a connection  
4 for which the form is formatted. This property has no effect on output. For a connec-  
5 tion that results from execution of an OPEN statement, the blank significance property  
6 is NULL by default if no blank significance property is specified. For a preconnected  
7 file, the property is NULL.
- 8 The blank significance property of the connection is effective at the beginning of each  
9 formatted input statement. During execution of the statement, any BN or BZ edit  
10 descriptors encountered may temporarily change the effect of embedded and trailing  
11 blanks (9.3.4.6).
- 12 FORTRAN 77 did not define default values for the blank significance properties of internal and pre-  
13 connected files. This standard defines the default values for these files to be NULL, matching that  
14 of files connected by the OPEN statement.
- 15 A processor has wide latitude in adapting these concepts and actions to its own cataloging and job  
16 control conventions. Some processors may require job control action to specify the set of files that  
17 exist or that will be created by an executable program. Some processors may require no job con-  
18 trol action prior to execution. This standard enables processors to perform dynamic open, close, or  
19 file creation operations, but it does not require such capabilities of the processor.
- 20 The meaning of "open" in contexts other than Fortran may include such things as mounting a  
21 tape, console messages, spooling, label checking, security checking, etc. These actions may occur  
22 upon job control action external to Fortran, upon execution of an OPEN statement, or upon execu-  
23 tion of the first read or write of the file. The OPEN statement describes properties of the connec-  
24 tion to the file and may or may not cause physical activities to take place. It is a place for an imple-  
25 mentation to define properties of a file beyond those required in standard Fortran.
- 26 **C.9.5 CLOSE Statement (9.3.5).** Similarly, the actions of dismounting a tape, protection, etc. of a  
27 "close" may be implicit at the end of a run. The CLOSE statement may or may not cause such  
28 actions to occur. This is another place to extend file properties beyond those of standard Fortran.  
29 Note, however, that the execution of a CLOSE statement on a unit followed by an OPEN statement  
30 on the same unit to the same file or to a different file is a permissible sequence of events. The proc-  
31 essor must not deny this sequence solely because the implementation chooses to do the physical  
32 act of closing the file at the termination of execution of the program.

**Section Notes**

1 **C.9.6 INQUIRE Statement (9.6).** Table C.2 indicates the values assigned to the INQUIRE state-  
 2 ment specifier variables when no error condition is encountered during execution of the INQUIRE  
 3 statement.

4 **Table C.2 Values Assigned to INQUIRE Specifier Variables**

| Specifier    | INQUIRE by File                                 |                                                                | INQUIRE by Unit                             |             |
|--------------|-------------------------------------------------|----------------------------------------------------------------|---------------------------------------------|-------------|
|              | Unconnected                                     | Connected                                                      | Connected                                   | Unconnected |
| EXIST=       | .TRUE. if file exists,<br>.FALSE. otherwise     |                                                                | .TRUE. if unit exists,<br>.FALSE. otherwise |             |
| OPENED=      | .FALSE.                                         | .TRUE.                                                         |                                             | .FALSE.     |
| NUMBER=      | -1                                              | unit no.                                                       |                                             | -1          |
| NAMED=       | .TRUE.                                          |                                                                | .TRUE. if file named,<br>.FALSE. otherwise  | .FALSE.     |
| NAME=        | filename<br>(may not be same<br>as FILE= value) |                                                                | filename<br>if named,<br>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;<br>else maximum record length |                                             | undefined   |
| NEXTREC=     | undefined                                       | if direct access, next record #;<br>else undefined             |                                             | undefined   |
| BLANK=       | UNDEFINED                                       | NULL, ZERO, or UNDEFINED                                       |                                             | UNDEFINED   |
| DELIM=       | UNDEFINED                                       | APOSTROPHE, QUOTE,<br>NONE, or UNDEFINED                       |                                             | UNDEFINED   |
| PAD=         | YES                                             | YES or NO                                                      |                                             | YES         |
| POSITION=    | UNDEFINED                                       | REWIND, APPEND,<br>ASIS, or UNDEFINED                          |                                             | UNDEFINED   |
| ACTION=      | UNDEFINED                                       | READ, WRITE,<br>or READWRITE                                   |                                             | UNDEFINED   |
| IOLENGTH=    | RECL= value for <i>output-item-list</i>         |                                                                |                                             |             |

60 **C.9.7 Keyword Specifiers.** Keyword forms of specifiers are used because there are many specifi-  
 61 ers and a positional notation is difficult to remember. The keyword form sets a style for processor  
 62 extensions. The UNIT= and FMT= keywords are offered for completeness, but their use is  
 63 optional. Thus, compatibility with ANSI X3.9-1966 (FORTRAN 66) and FORTRAN 77 is achieved.

64 **C.9.8 Format Specifications (9.4.1.1).** Format specifications may be included in the READ and  
 65 WRITE statements, as in:

```
66 READ (UNIT = 10, FMT = '(I3, A4, F10.2)') K, ALPH, X
```



1 **C.9.9 Unformatted Input/Output (9.4.4.1).** Unformatted input/output involving derived-  
 2 type list items forms the single exception to the rule that the appearance of an aggregate list item  
 3 (such as an array) is equivalent to the appearance of its expanded list of component parts. This  
 4 exception permits the processor greater latitude in improving efficiency or in matching the  
 5 processor-dependent sequence of values for a derived-type object to similar sequences for aggregate  
 6 objects used by means other than Fortran. However, formatted input/output of all list items  
 7 and unformatted input/output of list items other than those of derived types adhere to the above  
 8 rule.

9 **C.9.10 Input/Output Restrictions.** An example of a restriction on input/output statements (9.8)  
 10 is that an input statement must not specify that data are to be read from a printer.

11 **C.9.11 Pointers in an Input/Output List.** Data transfers always involve the movement of values  
 12 between a file and internal storage. A pointer as such cannot be read or written. A pointer may,  
 13 therefore, appear as an item in an input/output list if it is currently associated with a target that  
 14 can receive a value (input) or can deliver a value (output). A derived type object with one or more  
 15 pointer components must not appear as an item in an input/output list because the value of a  
 16 pointer component is the descriptor for a location in memory. As such, this has no processor-  
 17 independent representation.

18 **C.9.12 Derived Type Objects in an Input/Output List (9.4.2).** A component of a derived type  
 19 may be declared to be private (4.4, 5.1.2.2). A derived-type object must not appear in an  
 20 input/output list if any of its components or subobjects of any of its components have been  
 21 declared to be private and its derived type definition does not appear in the same module as the  
 22 data transfer statement.

23 In a formatted input/output statement, edit descriptors are associated with effective list items,  
 24 which are always scalar and of intrinsic type. The rules in 9.4.2 determine the set of effective list  
 25 items corresponding to each actual list item in the statement. These rules may have to be applied  
 26 repetitively until all of the effective list items are scalar items of intrinsic type.

## 27 C.10 Section 10 Notes.

28 **C.10.1 Character Constant Format Specification (10.1.2, 10.7.1).** If a character constant is  
 29 used as a format specifier in an input/output statement, care must be taken that the value of the  
 30 character constant is a valid format specification. In particular, if a format specification delimited  
 31 by apostrophes contains an apostrophe edit descriptor, two apostrophes must be written to delimit  
 32 the apostrophe edit descriptor and four apostrophes must be written for each apostrophe that  
 33 occurs within the apostrophe edit descriptor. For example, the text:

34 2 ISN'T 3

35 may be written by various combinations of output statements and format specifications:

36 WRITE (6, 100) 2, 3

37 100 FORMAT (1X, I1, 1X, 'ISN'T', 1X, I1)

38 WRITE (6, '(1X, I1, 1X, ''ISN''T'', 1X, I1)') 2, 3

39 WRITE (6, '(A)') ' 2 ISN'T 3'

40 Note that doubling of internal apostrophes usually may be avoided by using quotation marks to  
 41 delimit the format specification and doubling of internal quotation marks usually may be avoided  
 42 by using apostrophes as delimiters.

## Section Notes

1 **C.10.2 T Edit Descriptor (10.6.1.1).** The T edit descriptor includes the vertical spacing character  
2 (9.4.5) in lines that are to be printed. T1 specifies the vertical spacing character and T2 specifies the  
3 first character that is printed.

4 **C.10.3 Length of Formatted Records.** The length of a formatted record is not always specified  
5 exactly and may be processor dependent (10.8.2, 10.9.2).

6 **C.10.4 Number of Records (10.3, 10.4, 10.6.2).** The number of records read by an explicitly  
7 formatted advancing input statement can be determined from the following rule: a record is read  
8 at the beginning of the format scan (even if the input list is empty), at each slash edit descriptor  
9 encountered in the format, and when a format rescan occurs at the end of the format.

10 The number of records written by an explicitly formatted advancing output statement can be  
11 determined from the following rule: a record is written when a slash edit descriptor is encountered  
12 in the format, when a format rescan occurs at the end of the format, and at completion of execution  
13 of the output statement (even if the output list is empty). Thus, the occurrence of  $n$  successive  
14 slashes between two other edit descriptors causes  $n - 1$  blank lines if the records are printed. The  
15 occurrence of  $n$  slashes at the beginning or end of a complete format specification causes  $n$  blank  
16 lines if the records are printed. However, a complete format specification containing  $n$  slashes ( $n >$   
17  $0$ ) and no other edit descriptors causes  $n + 1$  blank lines if the records are printed. For example,  
18 the statements

```
19 PRINT 3  
20 3 FORMAT (/)
```

21 will write two records that cause two blank lines if the records are printed.

22 **C.10.5 List-Directed Input/Output (10.8).** List-directed input/output allows data editing  
23 according to the type of the list item instead of by a format specifier. It also allows data to be free-  
24 field, that is, separated by commas or blanks.

25 If no list items are specified in a list-directed input/output statement, one input record is skipped  
26 or one empty output record is written.

27 **C.10.6 List-Directed Input (10.8.1).** The following examples illustrate list-directed input. A  
28 blank character is represented by b.

29 Example 1:

30 Program:

```
31 J = 3  
32 READ *, I  
33 READ *, J
```

34 Sequential input file:

```
35 record 1: b1b,4bbbb  
36 record 2: ,2bbbbbbbb
```

37 Result: I = 1, J = 3.

38 Explanation: The second READ statement reads the second record. The initial comma in the  
39 record designates a null value; therefore, J is not redefined.

40 Example 2:

41 Program:

```
42 CHARACTER A *8, B *1  
43 READ *, A, B
```

44 Sequential input file:

1 record 1: 'bbbbbbbbb'  
 2 record 2: 'QXY'b'z'

3 Result: A = 'bbbbbbbbb', B = 'Q'

4 Explanation: In the first record, the rightmost apostrophe is interpreted as delimiting the constant  
 5 (it cannot be the first of a pair of embedded apostrophes representing a single apostrophe because  
 6 this would involve the prohibited "splitting" of the pair by the end of a record); therefore, A is set  
 7 to the character constant 'bbbbbbbbb'. The end of a record acts as a blank, which in this case is a  
 8 value separator because it occurs between two constants.

9 **C.10.7 Namelist List Items for Character Input (10.9.1.3).** Corresponding to a namelist input  
 10 list item of character data type, the character constant must be delimited either with apostrophes  
 11 or with quotes. The delimiter is required to avoid ambiguity between undelimited character con-  
 12 stants and object names. The value of the DELIM= specifier, if any, in the OPEN statement for an  
 13 external file is ignored during namelist input (9.3.4.9).

14 **C.10.8 Namelist Output Records (10.9.2.2).** Namelist output records produced with a DELIM=  
 15 specifier with a value of NONE and which contain a character constant may not be acceptable as  
 16 namelist input records.

## 17 C.11 Section 11 Notes.

18 **C.11.1 Main Program and Block Data Program Unit (11.1, 11.4).** The name of the main pro-  
 19 gram or of a block data program unit has no explicit use within the Fortran language. It is avail-  
 20 able for documentation and for possible use by a processor.

21 A processor may implement an unnamed main program or unnamed block data program unit <sup>by</sup>  
 22 assigning it a default name. However, this name must not conflict with any other global name in a  
 23 standard-conforming executable program. This might be done by making the default name one  
 24 which is not permitted in a standard-conforming program (for example, by including a character  
 25 not normally allowed in names) or by providing some external mechanism such that for any given  
 26 program the default name can be changed to one that is otherwise unused.

27 <sup>12.1</sup> **C.11.2 Examples of Host Association (11.2.2).** <sup>12.1.2.2.1</sup> The first two examples are valid examples of  
 28 host association. The third example is an invalid example of host association.

29 Example 1:

```
30 PROGRAM A
31 INTEGER I, J
32 ...
33 CONTAINS
34 SUBROUTINE B
35     INTEGER I ! Declaration of I hides
36               ! program A's declaration of I
37     ...
38     I = J     ! Use of variable J from program A
39               ! through host association
40 END SUBROUTINE B
41 END PROGRAM A
```

42 Example 2:

```
43 PROGRAM A
44     TYPE T
45     ...
46 END TYPE T
47 ...
```

## Section Notes

```
1 CONTAINS
2   SUBROUTINE B
3     IMPLICIT TYPE (T) (C) ! Refers to type T declared below
4                             ! in subroutine B, not type T
5                             ! declared above in program A
6     ...
7     TYPE T
8     ...
9     END TYPE T
10    ...
11  END SUBROUTINE B
12 END PROGRAM A
```

### Example 3:

```
14 PROGRAM Q
15   REAL (KIND = 1) :: C
16   ...
17 CONTAINS
18   SUBROUTINE R
19     REAL (KIND = KIND (C)) :: D ! Invalid declaration
20                                 ! See below
21     REAL (KIND = 2) :: C
22     ...
23   END SUBROUTINE R
24 END PROGRAM Q
```

25 In the declaration of D in subroutine R, the use of C would refer to the declaration of C in subrou-  
26 tine R, not program Q. However, it is invalid because the declaration of C must occur before it is  
27 used in the declaration of D.

28 **C.11.3 Dependent Compilation (11.3).** This standard, like its predecessors, is intended to per-  
29 mit the implementation of conforming processors in which a program can be broken into multiple  
30 units, each of which can be separately translated in preparation for execution. Such processors are  
31 commonly described as supporting separate compilation. There is an important difference  
32 between the way separate compilation can be implemented under this standard and the way it  
33 could be implemented under the previous standards. Under the previous standards, any informa-  
34 tion required to translate a program unit was specified in that program unit. Each translation was  
35 thus totally independent of all others. Under this standard, a program unit can use information  
36 that was specified in a separate module and thus may be dependent on that module. The imple-  
37 mentation of this dependency in a processor may be that the translation of a program unit may  
38 depend on the results of translating one or more modules. Processors implementing the depend-  
39 ency this way are commonly described as supporting dependent compilation.

40 The dependencies involved here are new only in the sense that the Fortran processor is now aware  
41 of them. The same information dependencies existed under the previous standards, but it was the  
42 programmer's responsibility to transport the information necessary to resolve them by making  
43 redundant specifications of the information in multiple program units. The availability of separate  
44 but dependent compilation offers several potential advantages over the redundant textual specifi-  
45 cation of information:

- 46 (1) Specifying information at a single place in the program ensures that different program  
47 units using that information will be translated consistently. Redundant specification  
48 leaves the possibility that different information will erroneously be specified. Even if  
49 some kind of textual inclusion facility is used to ensure that the text of the specifications  
50 is identical in all involved program units, the presence of other specifications (for exam-  
51 ple, an IMPLICIT statement) may change the interpretation of that text.
- 52 (2) During the revision of a program, it is possible for a processor to assist in determining  
53 whether different program units have been translated using different (incompatible)

- 1 versions of a module, although there is no requirement that a processor provide such  
 2 assistance. Inconsistencies in redundant textual specification of information, on the  
 3 other hand, tend to be much more difficult to detect.
- 4 (3) Putting information in a module provides a way of packaging it. Without modules,  
 5 redundant specifications frequently must be interleaved with other specifications in a  
 6 program unit, making convenient packaging of such information difficult.
- 7 (4) Because a processor may be implemented such that the specifications in a module are  
 8 translated once and then repeatedly referenced, there is the potential for greater effi-  
 9 ciency than when the processor must translate redundant specifications of information  
 10 in multiple program units.

11 The exact meaning of the requirement that the public portions of a module be available at the time  
 12 of reference is processor defined. For example, a processor could consider a module to be avail-  
 13 able only after it has been compiled and require that if the module has been compiled separately,  
 14 the result of that compilation must be identified to the compiler when compiling program units  
 15 that use it.

16 **C.11.3.1 USE Statement and Dependent Compilation (11.3.2).** Another benefit of the USE  
 17 statement is its enhanced facilities for name management. If one needs to use only selected entities  
 18 in a module, one can do so without having to worry about the names of all the other entities in that  
 19 module. If one needs to use two different modules that happen to contain entities with the same  
 20 name, there are several ways to deal with the conflict. If none of the entities with the same name  
 21 are to be used, they can simply be ignored. If the name happens to refer to the same entity in both  
 22 modules (for example, if both modules obtained it from a third module), then there is no confusion  
 23 about what the name denotes and the name can be freely used. If the entities are different and one  
 24 or both is to be used, the local renaming facility in the USE statement makes it possible to give  
 25 those entities different names in the program unit containing the USE statements.

26 A typical implementation of dependent but separate compilation may involve storing the result of  
 27 translating a module in a file (or file element) whose name is derived from the name of the mod-  
 28 ule. Note, however, that the name of a module is limited only by the Fortran rules and not by the  
 29 names allowed in the file system. Thus the processor may have to provide a mapping between  
 30 Fortran names and file system names.

31 The result of translating a module could reasonably either contain only the information textually  
 32 specified in the module (with "pointers" to information originally textually specified in other  
 33 modules) or contain all information specified in the module (including copies of information origi-  
 34 nally specified in other modules). Although the former approach would appear to save on storage  
 35 space, the latter approach can greatly simplify the logic necessary to process a USE statement and  
 36 can avoid the necessity of imposing a limit on the logical "nesting" of modules via the USE state-  
 37 ment.

38 Variables declared in a module retain their definition status on much the same basis as variables in  
 39 a common block. That is, saved variables retain their definition status throughout the execution of  
 40 a program, while variables that are not saved retain their definition status only during the execu-  
 41 tion of scoping units that reference the module. In some cases, it may be appropriate to put a USE  
 42 statement such as

43 `USE MY_MODULE, ONLY:`

44 in a scoping unit in order to assure that other procedures that it references can communicate  
 45 through the module. In such a case, the scoping unit would not access any entities from the mod-  
 46 ule, but the variables not saved in the module would retain their definition status throughout the  
 47 execution of the scoping unit.

48 There is an increased potential for undetected errors in a scoping unit that uses both implicit typ-  
 49 ing and the USE statement. For example, in the program fragment

## Section Notes

```
1 SUBROUTINE SUB
2   USE MY_MODULE
3   IMPLICIT INTEGER (I-N), REAL (A-H, O-Z)
4   X = F (B)
5   A = G (X) + H (X + 1)
6 END SUBROUTINE
```

7 X could be either an implicitly typed real variable or a variable obtained from the module  
8 MY\_MODULE and might change from one to the other because of changes in MY\_MODULE unre-  
9 lated to the action performed by SUB. Logic errors resulting from this kind of situation can be  
10 extremely difficult to locate. Thus, the use of these features together is discouraged.

11 **C.11.3.2 Accessibility Attributes (11.3.1).** The PUBLIC and PRIVATE attributes, which can be  
12 declared only in modules, divide the entities in a module into those which are actually relevant to  
13 a scoping unit referencing the module and those that are not. This information may be used to  
14 improve the performance of a Fortran processor. For example, it may be possible to discard much  
15 of the information on the private entities once a module has been translated, thus saving on both  
16 storage and the time to search it. Similarly, it may be possible to recognize that two versions of a  
17 module differ only in the private entities they contain and avoid retranslating program units that  
18 use that module when switching from one version of the module to the other.

19 **C.11.4 Pointers in Modules.** A pointer from a module program unit may be accessible in a pro-  
20 cedure via use association. Such pointers have a lifetime that is greater than targets that are  
21 declared in the procedure, unless such targets are saved. Therefore, if such a pointer is associated  
22 with a local target, there is the possibility that when the procedure completes execution, the target  
23 will cease to exist, leaving the pointer "dangling". This standard considers such pointers to be in  
24 an undefined state. They are neither associated nor disassociated. They must not be used again in  
25 the program until their status has been reestablished. There is no requirement on a processor to be  
26 able to detect when a pointer target ceases to exist.

27 **C.11.5 Example of a Module (11.3).** In addition to providing a portable means of avoiding the  
28 redundant specification of information in multiple program units, a module provides a convenient  
29 means of "packaging" related entities, such as the definitions of the representation and operations  
30 of an abstract data type. The following example of a module defines a data abstraction for a SET  
31 data type where the elements of each set are of type integer. The standard set operations of  
32 UNION, INTERSECTION, and DIFFERENCE are provided. The CARDINALITY function returns  
33 the cardinality of (number of elements in) its set argument. Two functions returning logical values  
34 are included, ELEMENT and SUBSET. ELEMENT ~~extends~~ *defines* the operator .IN. and SUBSET extends  
35 the operator <=. ELEMENT determines if a given scalar integer value is an element of a given set,  
36 and SUBSET determines if a given set is a subset of another given set. (Two sets may be checked  
37 for equality by comparing cardinality and checking that one is a subset of the other, or checking to  
38 see if each is a subset of the other.)

39 The transfer function SETF converts a vector of integer values to the corresponding set, with dupli-  
40 cate values removed. Thus, a vector of constant values can be used as set constants. An inverse  
41 transfer function VECTOR returns the elements of a set as a vector of values in ascending order. In  
42 this SET implementation, set data objects have a maximum ~~size (number of elements in set)~~ *cardinality* of 200.

```
43 MODULE INTEGER_SETS
44
45   INTEGER, PARAMETER :: MAX_SET_CARD = 200
46
47   TYPE SET
48     PRIVATE
49     INTEGER CARD
50     INTEGER ELEMENT (MAX_SET_CARD)
51   ! Define SET data type
52 END TYPE SET
```

```

1  INTERFACE OPERATOR (.IN.)
2      MODULE PROCEDURE ELEMENT
3  END INTERFACE

4  INTERFACE OPERATOR (<=)
5      MODULE PROCEDURE SUBSET
6  END INTERFACE

7  INTERFACE OPERATOR (+)
8      MODULE PROCEDURE UNION
9  END INTERFACE

10 INTERFACE OPERATOR (-)
11     MODULE PROCEDURE DIFFERENCE
12 END INTERFACE

13 INTERFACE OPERATOR (*)
14     MODULE PROCEDURE INTERSECTION
15 END INTERFACE

16 CONTAINS

17 INTEGER FUNCTION CARDINALITY (A)      ! Returns cardinality of set A
18     TYPE (SET) A
19     CARDINALITY = A % CARD
20 END FUNCTION CARDINALITY

21 LOGICAL FUNCTION ELEMENT (X, A)      ! Determines if
22     INTEGER X                        ! element X is in set A
23     TYPE (SET) A
24     ELEMENT = ANY (A % ELEMENT (1 : A % CARD) .EQ. X)
25 END FUNCTION ELEMENT

26 FUNCTION UNION (A, B)                ! Union of sets A and B
27     TYPE (SET) A, B, UNION
28     INTEGER J
29     UNION = A
30     DO J = 1, B % CARD
31         IF (.NOT. (B % ELEMENT (J) .IN. A)) THEN
32             IF (UNION % CARD < MAX_SET_CARD) THEN
33                 UNION % CARD = UNION % CARD + 1
34                 UNION % ELEMENT (UNION % CARD) = &
35                     B % ELEMENT (J)
36             ELSE
37                 ! Maximum set size exceeded . . .
38             END IF
39         END IF
40     END DO
41 END FUNCTION UNION

42 FUNCTION DIFFERENCE (A, B)           ! Difference of sets A and B
43     TYPE (SET) A, B, DIFFERENCE
44     INTEGER J, X
45     DIFFERENCE % CARD = 0             ! The empty set

```

## Section Notes

```
1     DO J = 1, A % CARD
2       X = A % ELEMENT (J)
3       IF (.NOT. (X .IN. B)) DIFFERENCE = DIFFERENCE + SET (1, (X) x (X))
4     END DO
5 END FUNCTION DIFFERENCE

6 FUNCTION INTERSECTION (A, B)           ! Intersection of sets A and B
7   TYPE (SET) A, B, INTERSECTION
8   INTERSECTION = A - (A - B)
9 END FUNCTION INTERSECTION

10 LOGICAL FUNCTION SUBSET (A, B)        ! Determines if set A is
11   TYPE (SET) A, B                    ! a subset of set B
12   INTEGER I
13   SUBSET = A % CARD <= B % CARD
14   IF (NOT SUBSET) RETURN THEN           ! For efficiency
15   DO I = 1, A % CARD
16     SUBSET = SUBSET .AND. (A % ELEMENT (I) .IN. B)
17   END DO END IF
18 END FUNCTION SUBSET

19 TYPE (SET) FUNCTION SETF (V)         ! Transfer function between a vector
20   INTEGER V (:)                     ! of elements and a set of elements
21   INTEGER J                           ! removing duplicate elements
22   SETF % CARD = 0
23   DO J = 1, SIZE (V)
24     IF (.NOT. (V (J) .IN. SETF)) THEN
25       IF (SETF % CARD < MAX_SET_CARD) THEN
26         SETF % CARD = SETF % CARD + 1
27         SETF % ELEMENT (SETF % CARD) = V (J)
28       ELSE
29         ! Maximum set size exceeded . . .
30       END IF
31     END IF
32   END DO
33 END FUNCTION SETF

34 FUNCTION VECTOR (A)                  ! Transfer the values of set A
35   TYPE (SET) A                       ! into a vector in ascending order
36   INTEGER, POINTER :: VECTOR (:)
37   INTEGER I, J, K
38   ALLOCATE (VECTOR (A % CARD))
39   VECTOR = A % ELEMENT (1 : A % CARD)
40   DO I = 1, A % CARD - 1             ! Use a better sort if
41     DO J = I + 1, A % CARD           ! A % CARD is large
42       IF (VECTOR (I) > VECTOR (J)) THEN
43         K = VECTOR (J); VECTOR (J) = VECTOR (I); VECTOR (I) = K
44       END IF
45     END DO
46   END DO
47 END FUNCTION VECTOR

48 END MODULE INTEGER_SETS

49 Examples of using INTEGER_SETS (A, B, and C are variables of type SET; X is an integer variable):
```



```

1  ! Check to see if A has more than 10 elements
2  IF (CARDINALITY (A) > 10) ...

3  ! Check for X an element of A but not of B
4  IF (X .IN. (A - B)) ...

5  ! C is the union of A and the result of B intersected
6  ! with the integers 1 to 100
7  C = A + B * SETF ((/ (I, I = 1, 100). /))

8  ! Does A have any even numbers in the range 1:100?
9  IF (CARDINALITY (A * SETF ((/ (I, I = 2, 100, 2) /)) > 0) ...

10 PRINT *, VECTOR (B) ! Print out the elements of set B, in ascending order

```

## 11 C.12 Section 12 Notes.

12 **C.12.1 External Procedures (12.3.2.2).** 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 ensure that none of these names conflicts with any of the names of the external procedures, with other globally named entities in a standard-conforming program, or with each other. For example, this might be done by including in each such added name a character that is not allowed in a standard-conforming name or by using such a character to combine a local designation with the global name of the program unit in which it appears.

22 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.

32 **C.12.2 Procedures Defined by Means Other Than Fortran (12.5.3).** A processor is not required to provide any means other than Fortran for defining external procedures. Among the means that might be supported are the machine assembly language, other high level languages, the Fortran language extended with nonstandard features, and the Fortran language as supported by another Fortran processor (for example, a previously existing FORTRAN 77 processor).

37 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.

46 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

## Section Notes

1 changed by a procedure reference unless that variable were one of the arguments to the procedure.

2 **C.12.3 Procedure Interfaces (12.3).** In FORTRAN 77, the interface to an external procedure was  
3 always deduced from the form of references to that procedure and any declarations of the proce-  
4 dure name in the referencing program unit. In this standard, features such as argument keywords  
5 and optional arguments make it impossible to deduce sufficient information about the dummy  
6 arguments from the nature of the actual arguments to be associated with them, and features such  
7 as array-valued function results and pointer function results make necessary extensions to the dec-  
8 laration of a procedure that cannot be done in a way that would be analogous with the handling of  
9 such declarations in FORTRAN 77. Hence, mechanisms are provided through which all the informa-  
10 tion about a procedure's interface may be made available in a scoping unit that references it. A  
11 procedure whose interface must be deduced as in FORTRAN 77 is described as having an implicit  
12 interface. A procedure whose interface is fully known is described as having an explicit interface.

13 A scoping unit is allowed to contain a procedure interface block for procedures that do not exist in  
14 the executable program, provided the procedure described is never referenced. The purpose of  
15 this rule is to allow implementations in which the use of a module providing procedure interface  
16 blocks describing the interface of every routine in a library would not automatically cause each of  
17 those library routines to be a part of the program referencing the module. Instead, only those  
18 library procedures actually referenced would be a part of the executable program. (In implemen-  
19 tation terms, the mere presence of a procedure interface block would not generate an external ref-  
20 erence in such an implementation.)

21 **C.12.4 Argument Association and Evaluation (12.4.1).** There is a significant difference  
22 between the argument association allowed in this standard and that supported by FORTRAN 77 and  
23 FORTRAN 66. In FORTRAN 77 and 66, actual arguments were limited to consecutive storage units.  
24 With the exception of assumed length character dummy arguments, the structure imposed on that  
25 sequence of storage units was always determined in the invoked procedure and not taken from the  
26 actual argument. Thus it was possible to implement FORTRAN 66 and FORTRAN 77 argument associa-  
27 tion by supplying only the location of the first storage unit (except for character arguments, where  
28 the length would also have to be supplied). However, this standard allows arguments that do not  
29 reside in consecutive storage locations (for example, an array section), and dummy arguments that  
30 assume additional structural information from the actual argument (for example, assumed-shape  
31 dummy arguments). Thus, the mechanism to implement the argument association allowed in this  
32 standard must be more general.

33 Because there are practical advantages to a processor that can support references to and from proce-  
34 dures defined by a FORTRAN 77 processor, requirements for explicit interfaces have been added to  
35 make it possible to determine whether a simple (FORTRAN 66/FORTRAN 77) argument association  
36 implementation mechanism is sufficient or whether the more general mechanism is necessary  
37 (12.3.1.1). Thus a processor can be implemented whose procedures expect the simple mechanism  
38 to be used whenever the procedure's interface is one which uses only FORTRAN 77 features and  
39 which expects the more general mechanism otherwise (for example, if there are assumed-shape or  
40 optional arguments). At the point of reference, the appropriate mechanism can be determined  
41 from the interface if it is explicit and can be assumed to be the simple mechanism if it is not. Note  
42 that if the simple mechanism is determined to be what the procedure expects, it may be necessary  
43 for the processor to allocate consecutive temporary storage for the actual argument, copy the  
44 actual argument to the temporary storage, reference the procedure using the temporary storage in  
45 place of the actual argument, copy the contents of temporary storage back to the actual argument,  
46 and deallocate the temporary storage.

47 Note that while this is the specific implementation method these rules were designed to support, it  
48 is not the only one possible. For example, on some processors, it may be possible to implement the  
49 general argument association in such a way that the information involved in FORTRAN 77 argument  
50 association may be found in the same places and the "extra" information is placed so it does not  
51 disturb a procedure expecting only FORTRAN 77 argument association. With such an implementa-  
52 tion, argument association could be translated without regard to whether the interface is explicit or  
53 implicit. Alternatively, it would be possible to disallow discontinuous arguments when calling  
54 procedures defined by the FORTRAN 77 processor and let any copying to and from contiguous

1 storage be done explicitly in the program. Yet another possibility would be not to allow references  
2 to procedures defined by a FORTRAN 77 processor.

3 The provisions for expression evaluation give the processor considerable flexibility for obtaining  
4 expression values in the most efficient way possible. This includes not evaluating an operand, for  
5 example, if the value of the expression can be determined otherwise (7.1.7.1). This flexibility  
6 applies to function argument evaluation, including the order of argument evaluation, delaying  
7 argument evaluation, and omitting argument evaluation. A processor may delay the evaluation of  
8 an argument in a procedure reference until the execution of the procedure refers to the value of  
9 that argument, provided delaying the evaluation of the argument does not otherwise affect the  
10 results of the executable program. The processor may, with similar restrictions, entirely omit the  
11 evaluation of an argument not referenced in the execution of the procedure. This gives processors  
12 latitude for optimization (for example, for parallel processing).

13 Note that successive commas must not be used to omit optional arguments.

14 **C.12.5 Argument Intent Specification (12.4.1.1).** Argument intent specifications serve several  
15 purposes in addition to documenting the intended use of dummy arguments. A processor can  
16 check whether an INTENT (IN) dummy argument is used in a way that could redefine it. A  
17 slightly more sophisticated processor could check to see whether an INTENT (OUT) dummy argu-  
18 ment could possibly be referenced before it is defined. If the procedure's interface is explicit, the  
19 processor can also verify that actual arguments corresponding to INTENT (OUT) or INTENT  
20 (INOUT) dummy arguments are definable. A more sophisticated processor could use this infor-  
21 mation to optimize the translation of the referencing scoping unit by taking advantage of the fact  
22 that actual arguments corresponding to INTENT (IN) dummy arguments will not be changed and  
23 that any prior value of an actual argument corresponding to an INTENT (OUT) dummy argument  
24 will not be referenced and can thus be discarded.

25 Note that INTENT (OUT) means that the value of the argument after invoking the procedure is  
26 entirely the result of executing that procedure. If there is any possibility that an argument should  
27 retain its current value rather than being redefined, INTENT (INOUT) should be used rather than  
28 INTENT (OUT), even if there is no explicit reference to the value of the dummy argument.

29 Note also that INTENT (INOUT) is not equivalent to the default. The argument corresponding to  
30 an INTENT (INOUT) dummy argument always must be definable, while an argument correspond-  
31 ing to a dummy argument with default intent need be definable only if the dummy argument is  
32 actually redefined.

33 **C.12.6 Dummy Argument Restrictions (12.5.2.9).** The restrictions on entities associated with  
34 dummy arguments are intended to allow a processor to translate a procedure on the assumption  
35 that each dummy argument is distinct from any other entity accessible in the procedure. This  
36 allows a variety of optimizations in the translation of the procedure, including implementations of  
37 argument association in which the value of the actual argument is maintained in a register or in  
38 local storage.

39 **C.12.7 Pointers and Targets as Arguments.** If a dummy argument is declared to be a pointer, it  
40 may be matched only by an actual argument that also is a pointer, and the characteristics of both  
41 arguments must agree. A model for such an association is that descriptor values of the actual  
42 pointer are copied to the dummy pointer. If the actual pointer has an associated target, this target  
43 becomes accessible via the dummy pointer. If the dummy pointer becomes associated with a dif-  
44 ferent target during execution of the procedure, this target will be accessible via the actual pointer  
45 after the procedure completes execution. If the dummy pointer becomes associated with a local  
46 target that ceases to exist when the procedure completes, the actual pointer will be left dangling in  
47 an undefined state. Such dangling pointers must not be used.

48 Since it is intended to allow implementations to make local copies of actual arguments, a pointer  
49 that is associated with a nonpointer actual argument is not associated with its corresponding  
50 dummy argument, since that latter might be stored in a local copy. Similarly, on return, a pointer  
51 that is associated with a nonpointer dummy argument becomes undefined, since the dummy

## Section Notes

1 argument may have been kept in a local copy that is no longer available. If it is required to main-  
2 tain such pointer association, the dummy argument should be replaced by a pointer to the target  
3 object. For example, consider the procedure

```
4 SUBROUTINE BEST (P, A)
5   REAL, POINTER :: P
6   REAL, TARGET  :: A (:)
7   ...                               ! Find the best element, A(I)
8   P => A (I)
9   RETURN
10  END
```

11 and the invocation

```
12 REAL, POINTER :: PBEST
13 REAL, TARGET  :: B (10000)
14 ...
15 CALL BEST (PBEST, B)
```

16 This leaves PBEST undefined. However, if A is given the attribute POINTER instead of TARGET:

```
17 SUBROUTINE BEST (P, A)
18   REAL, POINTER :: P
19   REAL, POINTER :: A (:)
20   ...                               ! Find the best element, A(I)
21   P => A (I)
22   RETURN
23  END
```

24 the invocation

```
25 REAL, POINTER :: PB, PBEST
26 REAL, TARGET  :: B (10000)
27 PB => B
28 ...                               ! Calculate B
29 CALL BEST (PBEST, PB)
```

30 leaves PBEST associated with the "best" element of B.

31 **C.12.8 The ASSOCIATED Function (13.13.13).** The ASSOCIATED intrinsic function may be used  
32 to test whether a pointer is associated with a target. The one-argument form is used for this pur-  
33 pose. In the two-argument form, the ASSOCIATED function tests whether the pointer first argu-  
34 ment is associated with the space that is referred to by the second argument. In most cases, it will  
35 be used to test if two pointers are associated with the same target.

36 **C.12.9 Internal Procedure Restrictions.** This standard does not allow internal procedures to be  
37 used as actual arguments, in part to simplify the problem of ensuring that internal procedures  
38 with recursive hosts access entities from the correct instance of the host. If, as an extension, a proc-  
39 essor allows internal procedures to be used as actual arguments, the correct instance in this case is  
40 the instance in which the procedure is supplied as an actual argument, even if the corresponding  
41 dummy argument is eventually invoked from a different instance.

42 **C.12.10 The Result Variable (12.5.2.2).** The result variable is similar to any other variable local  
43 to a function subprogram. Its existence begins when execution of the function is initiated and ends  
44 when execution of the function is terminated. However, because the final value of this variable is  
45 used subsequently in the evaluation of the expression that invoked the function, an implementa-  
46 tion may wish to defer releasing the storage occupied by that variable until after its value has been  
47 used in expression evaluation.

1 **C.13 Section 13 Notes.**2 **C.13.1 Summary of Features.** This section is a summary of the principal array features.3 **C.13.1.1 Whole Array Expressions and Assignments (7.5.1.2, 7.5.1.5).** An important new fea-  
4 ture is that whole array expressions and assignments are permitted. For example, the statement5 `A = B + C * SIN (D)`6 where A, B, C, and D are arrays of the same shape, is permitted. It is interpreted element-by-  
7 element; that is, the sine function is taken on each element of D, each result is multiplied by the  
8 corresponding element of C, added to the corresponding element of B, and assigned to the corre-  
9 sponding element of A. Functions, including user-written functions, may be array valued and  
10 may be generic with scalar versions. All arrays in an expression or across an assignment must con-  
11 form; that is, have exactly the same shape (number of dimensions and set of lengths in each dimen-  
12 sion), but scalars may be included freely and these are interpreted as being broadcast to a conform-  
13 ing array. Expressions are evaluated before any assignment takes place.14 **C.13.1.2 Array Sections (2.4.7, 6.2.2.3).** Whenever whole arrays may be used, it is also possible  
15 to use subarrays called "sections". For example:16 `A (:, 1:N, 2, 3:1:-1)`17 consists of a subarray containing the whole of the first dimension, positions 1 to N of the second  
18 dimension, position 2 of the third dimension and positions 1 to 3 in reverse order of the fourth  
19 dimension. This is an artificial example chosen to illustrate the different forms. Of course, the  
20 most common use is to select a row or column of an array, for example:21 `A (:, J)`22 **C.13.1.3 WHERE Statement (7.5.3).** The WHERE statement applies a conforming logical array as  
23 a mask on the individual operations in the expression and in the assignment. For example:24 `WHERE (A .GT. 0) B = LOG (A)`25 takes the logarithm only for positive components of A and makes assignments only in these posi-  
26 tions.

27 The WHERE statement also has a block form (WHERE construct).

28 **C.13.1.4 Automatic and Allocatable Arrays (5.1, 5.1.2.4.3).** A major advance for writing  
29 modular software is the presence of automatic arrays, created on entry to a subprogram and  
30 destroyed on return, and allocatable arrays whose rank is fixed but whose actual size and lifetime  
31 is fully under the programmer's control through explicit ALLOCATE and DEALLOCATE state-  
32 ments. The declarations33 `SUBROUTINE X (N, A, B)`34 `REAL WORK (N, N); REAL, ALLOCATABLE :: HEAP (:, :)`35 specify an automatic array WORK and an allocatable array HEAP. Note that a stack is an ade-  
36 quate storage mechanism for the implementation of automatic arrays, but a heap will be needed  
37 for allocatable arrays.38 **C.13.1.5 Array Constructors (4.5).** Arrays, and in particular array constants, may be constructed  
39 with array constructors exemplified by:40 `(/ 1.0, 3.0, 7.2 /)`

41 which is a rank-one array of size 3,

42 `(/ (1.3, 2.7, L = 1, 10), 7.1 /)`43 which is a rank-one array of size 21 and contains the pair of real constants 1.3 and 2.7 repeated 10  
44 times followed by 7.1, and

## Section Notes

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

2 which contains the integers 1, 2, ..., N. Only rank-one arrays may be constructed in this way, but  
3 higher dimensional arrays may be made from them by means of the intrinsic function RESHAPE.

4 **C.13.1.6 Intrinsic Functions.** All of the FORTRAN 77 intrinsic functions and all of the scalar intrinsic  
5 functions that have been added to the language have been extended to be applicable to arrays.  
6 Each such function is applied element-by-element to produce an array of the same shape. In addition,  
7 the following array intrinsics have been added, many of which return array-valued results.

### 8 C.13.1.6.1 Vector and Matrix Multiply Functions.

9 DOT\_PRODUCT (VECTOR\_A, VECTOR\_B) Dot product of two arrays  
10 MATMUL (MATRIX\_A, MATRIX\_B) Matrix multiplication

### 11 C.13.1.6.2 Array Reduction Functions.

12 ALL (MASK, DIM) True if all values are true  
13 ANY (MASK, DIM) True if any value is true  
14 COUNT (MASK, DIM) Number of true elements in an array  
15 MAXVAL (ARRAY, DIM, MASK) Maximum value in an array  
16 MINVAL (ARRAY, DIM, MASK) Minimum value in an array  
17 PRODUCT (ARRAY, DIM, MASK) Product of array elements  
18 SUM (ARRAY, DIM, MASK) Sum of array elements

### 19 C.13.1.6.3 Array Inquiry Functions.

20 ALLOCATED (ARRAY) Array allocation status  
21 LBOUND (ARRAY, DIM) Declared lower dimension bounds of an array  
22 SHAPE (SOURCE) Declared shape of an array or scalar  
23 SIZE (ARRAY, DIM) Declared total number of array elements  
24 UBOUND (ARRAY, DIM) Declared upper dimension bounds of an array

### 25 C.13.1.6.4 Array Construction Functions.

26 MERGE (TSOURCE, FSOURCE, MASK) Merge under mask  
27 PACK (ARRAY, MASK, VECTOR) Pack an array into a vector under a mask  
28 SPREAD (SOURCE, DIM, NCOPIES) Replicates an array by adding a dimension  
29 UNPACK (VECTOR, MASK, FIELD) Unpack a vector into an array under a mask

### 30 C.13.1.6.5 Array Reshape Function.

31 RESHAPE (SOURCE, SHAPE,  
32 PAD, ORDER) Reshape an array

### 33 C.13.1.6.6 Array Manipulation Functions.

34 CSHIFT (ARRAY, SHIFT, DIM) Circular shift  
35 EOSHIFT (ARRAY, SHIFT,  
36 BOUNDARY, DIM) End-off shift  
37 TRANSPOSE (MATRIX) Transpose of matrix

### 38 C.13.1.6.7 Array Location Functions.

39 MAXLOC (ARRAY, MASK) Location of a maximum value in an array  
40 MINLOC (ARRAY, MASK) Location of a minimum value in an array

41 **C.13.2 Examples.** The array features have the potential to simplify the way that almost any  
42 array-using program is conceived and written. Many algorithms involving arrays can now be  
43 written conveniently as a series of computations with whole arrays.

- 1 **C.13.2.1 Unconditional Array Computations.** At the simplest level, statements such as  
 2  $A = B + C$   
 3 or  
 4  $S = \text{SUM} (A)$   
 5 can take the place of entire DO loops. The loops were required to perform array addition or to  
 6 sum all the elements of an array.  
 7 Further examples of unconditional operations on arrays that are simple to write are:

matrix multiply       $P = \text{MATMUL} (Q, R)$   
 largest array element    $L = \text{MAXVAL} (P)$   
 factorial N               $F = \text{PRODUCT} ((/ (K, K = 2, N) /))$

- 11 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  
 12 use of the element-by-element definition of array expressions as described in Section 7. Thus, we  
 13 can write  
 14  $F = \text{SUM} (A * \text{COS} (X))$   
 15 The successive stages of calculation of F would then involve the arrays:

$A = (/ A (1), \dots, A (N) /)$   
 $X = (/ X (1), \dots, X (N) /)$   
 $\text{COS} (X) = (/ \text{COS} (X (1)), \dots, \text{COS} (X (N)) /)$   
 $A * \text{COS} (X) = (/ A (1) * \text{COS} (X (1)), \dots, A (N) * \text{COS} (X (N)) /)$

- 20 The final scalar result is obtained simply by summing the elements of the last of these arrays.  
 21 Thus, the processor is dealing with arrays at every step of the calculation.  
 22 **C.13.2.2 Conditional Array Computations.** Suppose we wish to compute the Fourier sum in  
 23 the above example, but to include only those terms  $a(i) \cos x(i)$  that satisfy the condition that the  
 24 coefficient  $a(i)$  is less than 0.01 in absolute value. More precisely, we are now interested in evalu-  
 25 ating the conditional Fourier sum

$$CF = \sum_{|a_i| < 0.01} a_i \times \cos x_i$$

- 28 where the index runs from 1 to N as before.  
 29 This can be done by using the MASK parameter of the SUM function, which restricts the summa-  
 30 tion of the elements of the array  $A * \text{COS} (X)$  to those elements that correspond to true elements of  
 31 MASK. Clearly, the mask required is the logical array expression  $\text{ABS} (A) .\text{LT.} 0.01$ . Note that the  
 32 stages of evaluation of this expression are:

$A = (/ A (1), \dots, A (N) /)$   
 $\text{ABS} (A) = (/ \text{ABS} (A (1)), \dots, \text{ABS} (A (N)) /)$   
 $\text{ABS} (A) .\text{LT.} 0.01 = (/ \text{ABS} (A (1)) .\text{LT.} 0.01, \dots, \text{ABS} (A (N)) .\text{LT.} 0.01 /)$

- 36 The conditional Fourier sum we arrive at is:  
 37  $CF = \text{SUM} (A * \text{COS} (X), \text{MASK} = \text{ABS} (A) .\text{LT.} 0.01)$   
 38 If the mask is all false, the value of CF is zero.

## Section Notes

1 The use of a mask to define a subset of an array is crucial to the action of the WHERE statement.  
2 Thus for example, to set an entire array to zero, we may write simply  $A = 0$ ; but to set only the  
3 negative elements to zero, we need to write the conditional assignment

```
4 WHERE (A .LT. 0) A = 0
```

5 The WHERE statement complements ordinary array assignment by providing array assignment to  
6 any subset of an array that can be restricted by a logical expression.

7 In the Ising model described below, the WHERE statement predominates in use over the ordinary  
8 array assignment statement.

9 **C.13.2.3 A Simple Program: The Ising Model.** The Ising model is a well-known Monte Carlo  
10 simulation in 3-dimensional Euclidean space which is useful in certain physical studies. We will  
11 consider in some detail how this might be programmed. The model may be described in terms of  
12 a logical array of shape N by N by N. Each gridpoint is a single logical variable which is to be  
13 interpreted as either an up-spin (true) or a down-spin (false).

14 The Ising model operates by passing through many successive states. The transition to the next  
15 state is governed by a local probabilistic process. At each transition, all gridpoints change state  
16 simultaneously. Every spin either flips to its opposite state or not according to a rule that depends  
17 only on the states of its 6 nearest neighbors in the surrounding grid. The neighbors of gridpoints  
18 on the boundary faces of the model cube are defined by assuming cubic periodicity. In effect, this  
19 extends the grid periodically by replicating it in all directions throughout space.

20 The rule states that a spin is flipped to its opposite parity for certain gridpoints where a mere 3 or  
21 fewer of the 6 nearest neighbors currently have the same parity as it does. Also, the flip is exe-  
22 cuted only with probability P (4), P (5), or P (6) if as many as 4, 5, or 6 of them have the same parity  
23 as it does. (The rule seems to promote neighborhood alignments that may presumably lead to  
24 equilibrium in the long run.)

25 **C.13.2.3.1 Problems To Be Solved.** Some of the programming problems that we will need to  
26 solve in order to translate the Ising model into Fortran statements using entire arrays are:

- 27 (1) Counting nearest neighbors that have the same spin;
- 28 (2) Providing an array-valued function to return an array of random numbers; and
- 29 (3) Determining which gridpoints are to be flipped.

30 **C.13.2.3.2 Solutions in Fortran.** The arrays needed are:

```
31 LOGICAL ISING (N, N, N), FLIPS (N, N, N)  
32 INTEGER ONES (N, N, N), COUNT (N, N, N)  
33 REAL THRESHOLD (N, N, N)
```

34 The array-valued function needed is:

```
35 FUNCTION RAND (N)  
36 REAL RAND (N, N, N)
```

37 The transition probabilities are specified in the array

```
38 REAL P (6)
```

39 The first task is to count the number of nearest neighbors of each gridpoint g that have the same  
40 spin as g.

41 Assuming that ISING is given to us, the statements

```
42 ONES = 0  
43 WHERE (ISING) ONES = 1
```

44 make the array ONES into an exact analog of ISING in which 1 stands for an up-spin and 0 for a  
45 down-spin.



1 The next array we construct, COUNT, will record for every gridpoint of ISING the number of  
 2 spins to be found among the 6 nearest neighbors of that gridpoint. COUNT will be computed by  
 3 adding together 6 arrays, one for each of the 6 relative positions in which a nearest neighbor is  
 4 found. Each of the 6 arrays is obtained from the ONES array by shifting the ONES array one place  
 5 circularly along one of its dimensions. This use of circular shifting imparts the cubic periodicity.

```
6 COUNT = CSHIFT (ONES, SHIFT = -1, DIM = 1) &
7       + CSHIFT (ONES, SHIFT = 1, DIM = 1) &
8       + CSHIFT (ONES, SHIFT = -1, DIM = 2) &
9       + CSHIFT (ONES, SHIFT = 1, DIM = 2) &
10      + CSHIFT (ONES, SHIFT = -1, DIM = 3) &
11      + CSHIFT (ONES, SHIFT = 1, DIM = 3)
```

12 At this point, COUNT contains the count of nearest neighbor up-spins even at the gridpoints  
 13 where the Ising model has a down-spin. But we want a count of down-spins at those gridpoints,  
 14 so we correct COUNT at the down (false) points of ISING by writing:

```
15 WHERE (.NOT. ISING) COUNT = 6 - COUNT
```

16 Our object now is to use these counts of what may be called the "like-minded nearest neighbors"  
 17 to decide which gridpoints are to be flipped. This decision will be recorded as the true elements of  
 18 an array FLIP. The decision to flip will be based on the use of uniformly distributed random num-  
 19 bers from the interval  $0 \leq p < 1$ . These will be provided at each gridpoint by the array-valued  
 20 function RAND. The flip will occur at a given point if and only if the random number at that point  
 21 is less than a certain threshold value. In particular, by making the threshold value equal to 1 at the  
 22 points where there are 3 or fewer like-minded nearest neighbors, we guarantee that a flip occurs  
 23 at those points (because  $p$  is always less than 1). Similarly, the threshold values corresponding to  
 24 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  
 25 flip at those points ( $P(4)$ ,  $P(5)$ , and  $P(6)$  are input parameters in the range 0 to 1).

26 The thresholds are established by the statements:

```
27 THRESHOLD = 1.0
28 WHERE (COUNT .EQ. 4) THRESHOLD = P (4)
29 WHERE (COUNT .EQ. 5) THRESHOLD = P (5)
30 WHERE (COUNT .EQ. 6) THRESHOLD = P (6)
```

31 and the spins that are to be flipped are located by the statement:

```
32 FLIPS = RAND (N) .LE. THRESHOLD
```

33 All that remains to complete one transition to the next state of the ISING model is to reverse the  
 34 spins in ISING wherever FLIPS is true:

```
35 WHERE (FLIPS) ISING = .NOT. ISING
```

36 **C.13.2.3.3 The Complete Fortran Subroutine.** The complete code, enclosed in a subroutine that  
 37 performs a sequence of transitions, is as follows:

```
38 SUBROUTINE TRANSITION (N, ISING, ITERATIONS, P)
39     LOGICAL ISING (N, N, N), FLIPS (N, N, N)
40     INTEGER ONES (N, N, N), COUNT (N, N, N)
41     REAL THRESHOLD (N, N, N), P (6)
42
43     DO I = 1, ITERATIONS
44         ONES = 0
45         WHERE (ISING) ONES = 1
46         COUNT = CSHIFT (ONES, -1, 1) + CSHIFT (ONES, 1, 1) &
47              + CSHIFT (ONES, -1, 2) + CSHIFT (ONES, 1, 2) &
48              + CSHIFT (ONES, -1, 3) + CSHIFT (ONES, 1, 3)
49         WHERE (.NOT. ISING) COUNT = 6 - COUNT
```

## Section Notes

```
1      THRESHOLD = 1.0
2      WHERE (COUNT .EQ. 4) THRESHOLD = P (4)
3      WHERE (COUNT .EQ. 5) THRESHOLD = P (5)
4      WHERE (COUNT .EQ. 6) THRESHOLD = P (6)
5      FLIPS = RAND (N) .LE. THRESHOLD
6      WHERE (FLIPS) ISING = .NOT. ISING
7      END DO
```

```
8 CONTAINS
9     FUNCTION RAND (N)
10      REAL RAND (N, N, N)
11      CALL RANDOM (HARVEST = RAND)
12      RETURN
13     END FUNCTION RAND
14 END
```

15 **C.13.2.3.4 Reduction of Storage.** The array ISING could be removed (at some loss of clarity) by  
16 representing the model in ONES all the time. The array FLIPS can be avoided by combining the  
17 two statements that use it as:

```
18 WHERE (RAND (N) .LE. THRESHOLD) ISING = .NOT. ISING
```

19 but an extra temporary array would probably be needed. Thus, the scope for saving storage while  
20 performing whole array operations is limited. If N is small, this will not matter and the use of  
21 whole array operations is likely to lead to good execution speed. If N is large, storage may be very  
22 important and adequate efficiency will probably be available by performing the operations plane  
23 by plane. The resulting code is not as elegant, but all the arrays except ISING will have size of  
24 order  $N^2$  instead of  $N^3$ .

25 **C.13.3 FORMula TRANslation and Array Processing.** Many mathematical formulas can be  
26 translated directly into Fortran by use of the array processing features.

27 We assume the following array declarations:

```
28 REAL X (N), A (M, N)
```

29 Some examples of mathematical formulas and corresponding Fortran expressions follow.

30 **C.13.3.1 A Sum of Products.** The expression

$$\sum_{j=1}^N \prod_{i=1}^M a_{ij}$$

33 can be formed using the Fortran expression

```
34 SUM (PRODUCT (A, DIM=1))
```

35 The argument DIM=1 means that the product is to be computed down each column of A. If A had  
36 the value:

$$\begin{bmatrix} B & C & D \\ E & F & G \end{bmatrix}$$

39 the result of this expression is  $BE + CF + DG$ .

40 **C.13.3.2 A Product of Sums.** The expression

$$\prod_{i=1}^M \sum_{j=1}^N a_{ij}$$

- 1 can be formed using the Fortran expression  
 2 `PRODUCT (SUM (A, DIM = 2))`  
 3 The argument `DIM = 2` means that the sum is to be computed along each row of A. If A had the  
 4 value

$$\begin{bmatrix} B & C & D \\ E & F & G \end{bmatrix}$$

- 7 the result of this expression is  $(B+C+D)(E+F+G)$ .  
 8 **C.13.3.3 Addition of Selected Elements.** The expression

$$\sum_{x_i > 0.0} x_i$$

- 11 can be formed using the Fortran expression  
 12 `SUM (X, MASK = X .GT. 0.0)`  
 13 The mask locates the positive elements of the array of rank one. If X has the vector value (0.0, -0.1,  
 14 0.2, 0.3, 0.2, -0.1, 0.0), the result of this expression is 0.7.

- 15 **C.13.4 Sum of Squared Residuals.** The expression

$$\sum_{i=1}^N (x_i - x_{\text{mean}})^2$$

- 18 can be formed using the Fortran statements  
 19 `XMEAN = SUM (X) / SIZE (X)`  
 20 `SS = SUM ((X - XMEAN) ** 2)`  
 21 Thus, SS is the sum of the squared residuals.

- 22 **C.13.5 Vector Norms: Infinity-Norm and One-Norm.** The infinity-norm of vector  $X = (X(1), \dots,$   
 23  $X(N))$  is defined as the largest of the numbers  $ABS(X(1)), \dots, ABS(X(N))$  and therefore has the  
 24 value `MAXVAL (ABS (X))`.

- 25 The one-norm of vector X is defined as the *sum* of the numbers  $ABS(X(1)), \dots, ABS(X(N))$  and  
 26 therefore has the value `SUM ( ABS (X))`.

- 27 **C.13.6 Matrix Norms: Infinity-Norm and One-Norm.** The infinity-norm of the matrix  $A =$   
 28  $(A(I, J))$  is the largest row-sum of the matrix  $ABS(A(I, J))$  and therefore has the value  
 29 `MAXVAL (SUM (ABS, DIM = 2))`.

- 30 The one-norm of the matrix  $A = (A(I, J))$  is the largest column-sum of the matrix  $ABS(A(I, J))$  and  
 31 therefore has the value `MAXVAL (SUM (ABS (A), DIM = 1))`.

- 32 **C.13.7 Logical Queries.** The intrinsic functions allow quite complicated questions about tabular  
 33 data to be answered without use of loops or conditional constructs. Consider, for example, the  
 34 questions asked below about a simple tabulation of students' test scores.

- 35 Suppose the rectangular table T (M, N) contains the test scores of M students who have taken N  
 36 different tests. T is an integer matrix with entries in the range 0 to 100.

- 37 Example: The scores on 4 tests made by 3 students are held as the table

$$T = \begin{bmatrix} 85 & 76 & 90 & 60 \\ 71 & 45 & 50 & 80 \\ 66 & 45 & 21 & 55 \end{bmatrix}$$

- 2 Question: What is each student's top score?  
 3 Answer: MAXVAL (T, DIM = 2); in the example: [90, 80, 66].  
 4 Question: What is the average of all the scores?  
 5 Answer: SUM (T) / SIZE (T); in the example: 62.  
 6 Question: How many of the scores in the table are above average?  
 7 Answer: ABOVE = T .GT. SUM (T) / SIZE (T); N = COUNT (ABOVE); in the example: ABOVE is  
 8 the logical array (t = true, . = false):

$$\begin{bmatrix} t & t & t & . \\ t & . & . & t \\ t & . & . & . \end{bmatrix}$$

- 11 and COUNT (ABOVE) is 6.  
 12 Question: What was the lowest score in the above-average group of scores?  
 13 Answer: MINVAL (T, MASK = ABOVE), where ABOVE is as defined previously; in the example:  
 14 66.  
 15 Question: Was there a student whose scores were all above average?  
 16 Answer: With ABOVE as previously defined, the answer is yes or no according as the value of the  
 17 expression ANY (ALL (ABOVE, DIM = 2)) is true or false; in the example, the answer is no.  
 18 **C.13.8 Parallel Computations.** The most straightforward kind of parallel processing is to do the  
 19 same thing at the same time to many operands. Matrix addition is a good example of this very  
 20 simple form of parallel processing. Thus, the array assignment  $A = B + C$  specifies that corre-  
 21 sponding elements of the identically-shaped arrays B and C be added together in parallel and that  
 22 the resulting sums be assigned in parallel to the array A.  
 23 The process being done in parallel in the example of matrix addition is of course the process of  
 24 addition; the array feature that implements matrix addition as a parallel process is the element-  
 25 by-element evaluation of array expressions.  
 26 These observations lead us to look to element-by-element computation as a means of implement-  
 27 ing other simple parallel processing algorithms.  
 28 **C.13.9 Example of Element-by-Element Computation.** Several polynomials of the same  
 29 degree may be evaluated at the same point by arranging their coefficients as the rows of a matrix  
 30 and applying Horner's method for polynomial evaluation to the columns of the matrix so formed.  
 31 The procedure is illustrated by the code to evaluate the three cubic polynomials

$$P(t) = 1 + 2t - 3t^2 + 4t^3$$

$$Q(t) = 2 - 3t + 4t^2 - 5t^3$$

$$R(t) = 3 + 4t - 5t^2 + 6t^3$$

- 38 in parallel at the point  $t = X$  and to place the resulting vector of numbers [P (X), Q (X), R (X)] in the  
 39 real array RESULT (3).

- 1 The code to compute RESULT is just the one statement  
 2 `RESULT = M (:, 1) + X * (M (:, 2) + X * (M (:, 3) + X * M (:, 4)))`  
 3 where M represents the matrix M (3, 4) with value

$$\begin{bmatrix} 1 & 2 & -3 & 4 \\ 2 & -3 & 4 & -5 \\ 3 & 4 & -5 & 6 \end{bmatrix}$$

- 6 **C.13.10 Bit Manipulation and Inquiry Procedures.** The procedures IOR, IAND, NOT, IEOR,  
 7 ISHFT, ISHFTC, IBITS, MVBITS, BTEST, IBSET, and IBCLR are defined by MIL-STD 1753 for scalar  
 8 arguments and are extended in this standard to accept array arguments and to return array-valued  
 9 results.

## 10 C.14 Section 14 Notes.

- 11 **C.14.1 Unambiguous Generic Procedure References (14.1.2.3).** When multiple procedures  
 12 are associated with the same generic name, the same generic operator, or a defined assignment,  
 13 they must be distinguishable by their dummy argument lists. During the processing of a generic  
 14 interface block, the rules of 14.1.2.3 are applied to verify that the involved procedures are always  
 15 distinguishable.

- 16 There are different rules for generic operators and defined assignment than for generic names.  
 17 The difference involves the role of dummy argument rank in distinguishing procedures. In the  
 18 case of a generic name, the rank of a scalar, a pointer, or an assumed-shape array dummy argu-  
 19 ment is available to distinguish procedures. For generic operators and defined assignment, the  
 20 rank of any dummy argument is available.

- 21 As an example, consider the following interface block:

```
22 INTERFACE OPERATOR (+)
23   FUNCTION PROCESS_1 (A)
24     LOGICAL A
25   END
26   FUNCTION PROCESS_2 (B)
27     LOGICAL, POINTER :: B(:)
28   END
29   FUNCTION PROCESS_3 (C)
30     LOGICAL C (:, :)
31   END
32 END INTERFACE
```

- 33 In this example, the procedures are distinguishable under the rules of 14.1.2.3. However, errors  
 34 would result if the following two interface bodies are added:

```
35   FUNCTION PROCESS_4 (D)
36     LOGICAL D (10)
37   END
38   FUNCTION PROCESS_5 (E)
39     LOGICAL E (10, 10)
40   END
```

- 41 For PROCESS\_4, the dummy argument is not sufficiently distinct from that of PROCESS\_2. For  
 42 PROCESS\_5, the dummy argument is not sufficiently distinct from that of PROCESS\_3.

- 43 As a second example, suppose that the preceding interface block is for a generic name rather than  
 44 a generic operator. As before, the interface block is valid, and adding PROCESS\_4 and PROC-  
 45 ESS\_5 will again cause errors. In this case, the rank is used only for scalars, pointers, and  
 46 assumed-shape arrays, resulting in different errors. PROCESS\_4 and PROCESS\_5 each clash with

## Section Notes

1 all of PROCESS\_1, PROCESS\_2, and PROCESS\_3. In addition, PROCESS\_4 and PROCESS\_5 now  
2 will clash with each other.  
3 Enforcing these rules when processing an interface block will ensure that any generic reference  
4 will be unambiguous. However, note that stricter rules apply when interpreting a particular refer-  
5 ence to a generic operator (7.3) and defined assignment (7.5.1.6).

6 **C.14.2 Storage Association of Zero-Sized Objects.** Zero-sized objects may occur in a storage  
7 association context as the result of changing a parameter. For example, a program might contain  
8 the following declarations:

```
9 INTEGER, PARAMETER :: PROBSIZE = 10  
10 INTEGER, PARAMETER :: ARRAYSIZE = PROBSIZE * 100  
11 REAL, DIMENSION (ARRAYSIZE) :: X  
12 INTEGER, DIMENSION (ARRAYSIZE) :: IX  
13 ...  
14 COMMON / EXAMPLE / A, B, C, X, Y, Z  
15 EQUIVALENCE (X, IX)  
16 ...
```

17 If the first statement is subsequently changed to set PROBSIZE to zero, the program still will con-  
18 form to the standard.

## ANNEX D. SYNTAX RULES

1 (This Annex is not part of this standard, but is included for information only.)

2 This annex contains two parts. The first part is an extraction of all syntax rules and constraints in  
3 the order in which they appear in the standard. The second part is a cross reference with an entry  
4 for each terminal symbol and each nonterminal symbol in the syntax rules. The symbols are sorted  
5 alphabetically within three categories: nonterminal symbols that are defined, nonterminal symbols  
6 that are not defined, and terminal symbols.

7 Except for those ending with *-name*, the only undefined nonterminal symbols are *letter*, *digit*,  
8 *special-character*, and *rep-char*. As described in 1.5.2, the following rules are assumed. The letters  
9 "*xyz*" stand for any legal syntactic class phrase:

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

11 *xyz-name* is *name*

12 *scalar-xyz* is *xyz*

13 Constraint: *scalar-xyz* must be scalar.

14 **D.1 Syntax Rules and Constraints.** Each of the following sections contains the syntax rules  
15 and constraints from one section of the standard; that is, Section D.1.1 contains the rules and con-  
16 straints from Section 1 of the standard, D.1.2 contains those from Section 2, and so on. Note that  
17 Sections 1, 13, and 14 contain no syntax rules.

18 **D.1.1 Scope of Standard.**

19 **D.1.2 Fortran Terms and Concepts.**

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

22 R202 *program-unit* is *main-program*  
23 or *external-subprogram*  
24 or *module*  
25 or *block-data*

26 R1101 *main-program* is [ *program-stmt* ]  
27 [ *specification-part* ]  
28 [ *execution-part* ]  
29 [ *internal-subprogram-part* ]  
30 *end-program-stmt*

31 R203 *external-subprogram* is *function-subprogram*  
32 or *subroutine-subprogram*

33 R1215 *function-subprogram* is *function-stmt*  
34 [ *specification-part* ]  
35 [ *execution-part* ]  
36 [ *internal-subprogram-part* ]  
37 *end-function-stmt*

38 R1219 *subroutine-subprogram* is *subroutine-stmt*  
39 [ *specification-part* ]  
40 [ *execution-part* ]  
41 [ *internal-subprogram-part* ]  
42 *end-subroutine-stmt*

43 R1104 *module* is *module-stmt*  
44 [ *specification-part* ]  
45 [ *module-subprogram-part* ]  
46 *end-module-stmt*

**SYNTAX RULES**

|    |       |                                 |                                         |
|----|-------|---------------------------------|-----------------------------------------|
| 1  | R1110 | <i>block-data</i>               | is <i>block-data-stmt</i>               |
| 2  |       |                                 | [ <i>specification-part</i> ]           |
| 3  |       |                                 | <i>end-block-data-stmt</i>              |
| 4  | R204  | <i>specification-part</i>       | is [ <i>use-stmt</i> ] ...              |
| 5  |       |                                 | [ <i>implicit-part</i> ]                |
| 6  |       |                                 | [ <i>declaration-construct</i> ] ...    |
| 7  | R205  | <i>implicit-part</i>            | is [ <i>implicit-part-stmt</i> ] ...    |
| 8  |       |                                 | <i>implicit-stmt</i>                    |
| 9  | R206  | <i>implicit-part-stmt</i>       | is <i>implicit-stmt</i>                 |
| 10 |       |                                 | or <i>parameter-stmt</i>                |
| 11 |       |                                 | or <i>format-stmt</i>                   |
| 12 |       |                                 | or <i>entry-stmt</i>                    |
| 13 | R207  | <i>declaration-construct</i>    | is <i>derived-type-def</i>              |
| 14 |       |                                 | or <i>interface-block</i>               |
| 15 |       |                                 | or <i>type-declaration-stmt</i>         |
| 16 |       |                                 | or <i>specification-stmt</i>            |
| 17 |       |                                 | or <i>parameter-stmt</i>                |
| 18 |       |                                 | or <i>format-stmt</i>                   |
| 19 |       |                                 | or <i>entry-stmt</i>                    |
| 20 |       |                                 | or <i>stmt-function-stmt</i>            |
| 21 | R208  | <i>execution-part</i>           | is <i>executable-construct</i>          |
| 22 |       |                                 | [ <i>execution-part-construct</i> ] ... |
| 23 | R209  | <i>execution-part-construct</i> | is <i>executable-construct</i>          |
| 24 |       |                                 | or <i>format-stmt</i>                   |
| 25 |       |                                 | or <i>data-stmt</i>                     |
| 26 |       |                                 | or <i>entry-stmt</i>                    |
| 27 | R210  | <i>internal-subprogram-part</i> | is <i>contains-stmt</i>                 |
| 28 |       |                                 | <i>internal-subprogram</i>              |
| 29 |       |                                 | [ <i>internal-subprogram</i> ] ...      |
| 30 | R211  | <i>internal-subprogram</i>      | is <i>function-subprogram</i>           |
| 31 |       |                                 | or <i>subroutine-subprogram</i>         |
| 32 | R212  | <i>module-subprogram-part</i>   | is <i>contains-stmt</i>                 |
| 33 |       |                                 | <i>module-subprogram</i>                |
| 34 |       |                                 | [ <i>module-subprogram</i> ] ...        |
| 35 | R213  | <i>module-subprogram</i>        | is <i>function-subprogram</i>           |
| 36 |       |                                 | or <i>subroutine-subprogram</i>         |
| 37 | R214  | <i>specification-stmt</i>       | is <i>access-stmt</i>                   |
| 38 |       |                                 | or <i>allocatable-stmt</i>              |
| 39 |       |                                 | or <i>common-stmt</i>                   |
| 40 |       |                                 | or <i>data-stmt</i>                     |
| 41 |       |                                 | or <i>dimension-stmt</i>                |
| 42 |       |                                 | or <i>equivalence-stmt</i>              |
| 43 |       |                                 | or <i>external-stmt</i>                 |
| 44 |       |                                 | or <i>intent-stmt</i>                   |
| 45 |       |                                 | or <i>intrinsic-stmt</i>                |
| 46 |       |                                 | or <i>namelist-stmt</i>                 |
| 47 |       |                                 | or <i>optional-stmt</i>                 |
| 48 |       |                                 | or <i>pointer-stmt</i>                  |
| 49 |       |                                 | or <i>save-stmt</i>                     |
| 50 |       |                                 | or <i>target-stmt</i>                   |
| 51 | R215  | <i>executable-construct</i>     | is <i>action-stmt</i>                   |



|    |                         |                                                                                                                |
|----|-------------------------|----------------------------------------------------------------------------------------------------------------|
| 1  |                         | or <i>case-construct</i>                                                                                       |
| 2  |                         | or <i>do-construct</i>                                                                                         |
| 3  |                         | or <i>if-construct</i>                                                                                         |
| 4  |                         | or <i>where-construct</i>                                                                                      |
| 5  | R216 <i>action-stmt</i> | is <i>allocate-stmt</i>                                                                                        |
| 6  |                         | or <i>assignment-stmt</i>                                                                                      |
| 7  |                         | or <i>backspace-stmt</i>                                                                                       |
| 8  |                         | or <i>call-stmt</i>                                                                                            |
| 9  |                         | or <i>close-stmt</i>                                                                                           |
| 10 |                         | or <i>computed-goto-stmt</i>                                                                                   |
| 11 |                         | or <i>continue-stmt</i>                                                                                        |
| 12 |                         | or <i>cycle-stmt</i>                                                                                           |
| 13 |                         | or <i>deallocate-stmt</i>                                                                                      |
| 14 |                         | or <i>endfile-stmt</i>                                                                                         |
| 15 |                         | or <i>end-function-stmt</i>                                                                                    |
| 16 |                         | or <i>end-program-stmt</i>                                                                                     |
| 17 |                         | or <i>end-subroutine-stmt</i>                                                                                  |
| 18 |                         | or <i>exit-stmt</i>                                                                                            |
| 19 |                         | or <i>goto-stmt</i>                                                                                            |
| 20 |                         | or <i>if-stmt</i>                                                                                              |
| 21 |                         | or <i>inquire-stmt</i>                                                                                         |
| 22 |                         | or <i>nullify-stmt</i>                                                                                         |
| 23 |                         | or <i>open-stmt</i>                                                                                            |
| 24 |                         | or <i>pointer-assignment-stmt</i>                                                                              |
| 25 |                         | or <i>print-stmt</i>                                                                                           |
| 26 |                         | or <i>read-stmt</i>                                                                                            |
| 27 |                         | or <i>return-stmt</i>                                                                                          |
| 28 |                         | or <i>rewind-stmt</i>                                                                                          |
| 29 |                         | or <i>stop-stmt</i>                                                                                            |
| 30 |                         | or <i>where-stmt</i>                                                                                           |
| 31 |                         | or <i>write-stmt</i>                                                                                           |
| 32 |                         | or <i>arithmetic-if-stmt</i>                                                                                   |
| 33 |                         | or <i>assign-stmt</i>                                                                                          |
| 34 |                         | or <i>assigned-goto-stmt</i>                                                                                   |
| 35 |                         | or <i>pause-stmt</i>                                                                                           |
| 36 | Constraint:             | An <i>execution-part-construct</i> must not contain an <i>end-function-stmt</i> , <i>end-program-stmt</i> , or |
| 37 |                         | <i>end-subroutine-stmt</i> .                                                                                   |

### 38 D.1.3 Characters, Lexical Tokens, and Source Form.

|    |                                    |                                                        |
|----|------------------------------------|--------------------------------------------------------|
| 39 | R301 <i>character</i>              | is <i>alphanumeric-character</i>                       |
| 40 |                                    | or <i>special-character</i>                            |
| 41 | R302 <i>alphanumeric-character</i> | is <i>letter</i>                                       |
| 42 | <u>        </u>                    | or <i>digit</i>                                        |
| 43 | <u>        </u>                    | or <i>underscore</i>                                   |
| 44 | R303 <i>underscore</i>             | is <u>        </u>                                     |
| 45 | R304 <i>name</i>                   | is <i>letter</i> [ <i>alphanumeric-character</i> ] ... |
| 46 | Constraint:                        | The maximum length of a <i>name</i> is 31 characters.  |
| 47 | R305 <i>constant</i>               | is <i>literal-constant</i>                             |
| 48 |                                    | or <i>named-constant</i>                               |
| 49 | R306 <i>literal-constant</i>       | is <i>int-literal-constant</i>                         |
| 50 |                                    | or <i>real-literal-constant</i>                        |
| 51 |                                    | or <i>complex-literal-constant</i>                     |

## SYNTAX RULES

|    |                                                             |                                    |
|----|-------------------------------------------------------------|------------------------------------|
| 1  |                                                             | or <i>logical-literal-constant</i> |
| 2  |                                                             | or <i>char-literal-constant</i>    |
| 3  |                                                             | or <i>boz-literal-constant</i>     |
| 4  | R307 <i>named-constant</i>                                  | is <i>name</i>                     |
| 5  | R308 <i>int-constant</i>                                    | is <i>constant</i>                 |
| 6  | Constraint: <i>int-constant</i> must be of type integer.    |                                    |
| 7  | R309 <i>char-constant</i>                                   | is <i>constant</i>                 |
| 8  | Constraint: <i>char-constant</i> must be of type character. |                                    |
| 9  | R310 <i>intrinsic-operator</i>                              | is <i>power-op</i>                 |
| 10 |                                                             | or <i>mult-op</i>                  |
| 11 |                                                             | or <i>add-op</i>                   |
| 12 |                                                             | or <i>concat-op</i>                |
| 13 |                                                             | or <i>rel-op</i>                   |
| 14 |                                                             | or <i>not-op</i>                   |
| 15 |                                                             | or <i>and-op</i>                   |
| 16 |                                                             | or <i>or-op</i>                    |
| 17 |                                                             | or <i>equi-op</i>                  |
| 18 | R708 <i>power-op</i>                                        | is **                              |
| 19 | R709 <i>mult-op</i>                                         | is *                               |
| 20 |                                                             | or /                               |
| 21 | R710 <i>add-op</i>                                          | is +                               |
| 22 |                                                             | or -                               |
| 23 | R712 <i>concat-op</i>                                       | is //                              |
| 24 | R714 <i>rel-op</i>                                          | is .EQ.                            |
| 25 |                                                             | or .NE.                            |
| 26 |                                                             | or .LT.                            |
| 27 |                                                             | or .LE.                            |
| 28 |                                                             | or .GT.                            |
| 29 |                                                             | or .GE.                            |
| 30 |                                                             | or ==                              |
| 31 |                                                             | or /=                              |
| 32 |                                                             | or <                               |
| 33 |                                                             | or <=                              |
| 34 |                                                             | or >                               |
| 35 |                                                             | or >=                              |
| 36 | R719 <i>not-op</i>                                          | is .NOT.                           |
| 37 | R720 <i>and-op</i>                                          | is .AND.                           |
| 38 | R721 <i>or-op</i>                                           | is .OR.                            |
| 39 | R722 <i>equi-op</i>                                         | is .EQV.                           |
| 40 |                                                             | or .NEQV.                          |
| 41 | R311 <i>defined-operator</i>                                | is <i>defined-unary-op</i>         |
| 42 |                                                             | or <i>defined-binary-op</i>        |
| 43 |                                                             | or <i>extended-intrinsic-op</i>    |
| 44 | R704 <i>defined-unary-op</i>                                | is . letter [ letter ] ... .       |
| 45 | R724 <i>defined-binary-op</i>                               | is . letter [ letter ] ... .       |
| 46 | R312 <i>extended-intrinsic-op</i>                           | is <i>intrinsic-operator</i>       |

- 1 Constraint: A *defined-unary-op* and a *defined-binary-op* must not contain more than 31 letters and  
2 must not be the same as any *intrinsic-operator* or *logical-literal-constant*.
- 3 R313 *label* is *digit* [ *digit* [ *digit* [ *digit* ] ] ] ]
- 4 Constraint: At least one digit in a *label* must be nonzero.
- 5 **D.1.4 Intrinsic and Derived Data Types.**
- 6 R401 *signed-digit-string* is [ *sign* ] *digit-string*
- 7 R402 *digit-string* is *digit* [ *digit* ] ...
- 8 R403 *signed-int-literal-constant* is [ *sign* ] *int-literal-constant*
- 9 R404 *int-literal-constant* is *digit-string* [ *\_kind-param* ]
- 10 R405 *kind-param* is *digit-string*  
11 or *scalar-int-constant-name*
- 12 R406 *sign* is +  
13 or -
- 14 Constraint: The value of *kind-param* must be nonnegative.
- 15 Constraint: The value of *kind-param* must specify a representation method that exists on the proc-  
16 essor.
- 17 R407 *boz-literal-constant* is *binary-constant*  
18 or *octal-constant*  
19 or *hex-constant*
- 20 Constraint: A *boz-literal-constant* may appear only in a DATA statement.
- 21 R408 *binary-constant* is B' *digit* [ *digit* ] ... '  
22 or B" *digit* [ *digit* ] ... "
- 23 Constraint: *digit* must have one of the values 0 or 1.
- 24 R409 *octal-constant* is O' *digit* [ *digit* ] ... '  
25 or O" *digit* [ *digit* ] ... "
- 26 Constraint: *digit* must have one of the values 0 through 7.
- 27 R410 *hex-constant* is Z' *hex-digit* [ *hex-digit* ] ... '  
28 or Z" *hex-digit* [ *hex-digit* ] ... "
- 29 R411 *hex-digit* is *digit*  
30 or A  
31 or B  
32 or C  
33 or D  
34 or E  
35 or F
- 36 R412 *signed-real-literal-constant* is [ *sign* ] *real-literal-constant*
- 37 R413 *real-literal-constant* is *significand* [ *exponent-letter exponent* ] [ *\_kind-param* ]  
38 or *digit-string exponent-letter exponent* [ *\_kind-param* ]
- 39 R414 *significand* is *digit-string* . [ *digit-string* ]  
40 or . *digit-string*
- 41 R415 *exponent-letter* is E  
42 or D
- 43 R416 *exponent* is *signed-digit-string*

## SYNTAX RULES

- 1 Constraint: If both *kind-param* and *exponent-letter* are present, *exponent-letter* must be E.
- 2 Constraint: The value of *kind-param* must specify an approximation method that exists on the  
3 processor.
- 4 R417 *complex-literal-constant* is ( *real-part* , *imag-part* )
- 5 R418 *real-part* is *signed-int-literal-constant*  
6 or *signed-real-literal-constant*
- 7 R419 *imag-part* is *signed-int-literal-constant*  
8 or *signed-real-literal-constant*
- 9 R420 *char-literal-constant* is [ *kind-param* \_ ] ' [ *rep-char* ] ... '  
10 or [ *kind-param* \_ ] " [ *rep-char* ] ... "
- 11 Constraint: The value of *kind-param* must specify a representation method that exists on the proc-  
12 essor.
- 13 R421 *logical-literal-constant* is .TRUE. [ \_ *kind-param* ]  
14 or .FALSE. [ \_ *kind-param* ]
- 15 Constraint: The value of *kind-param* must specify a representation method that exists on the proc-  
16 essor.
- 17 R422 *derived-type-def* is *derived-type-stmt*  
18 [ *private-sequence-stmt* ] ...  
19 *component-def-stmt*  
20 [ *component-def-stmt* ] ...  
21 *end-type-stmt*
- 22 R423 *private-sequence-stmt* is PRIVATE  
23 or SEQUENCE
- 24 R424 *derived-type-stmt* is TYPE [ [ , *access-spec* ] :: ] *type-name*
- 25 Constraint: The same *private-sequence-stmt* must not appear more than once in a given *derived-*  
26 *type-def*.
- 27 Constraint: If SEQUENCE is present, all derived types specified in component definitions must  
28 be sequence types.
- 29 Constraint: An *access-spec* (5.1.2.2) or a PRIVATE statement within the definition is permitted  
30 only if the type definition is within the specification part of a module.
- 31 Constraint: If a component of a derived type is of a type declared to be private, either the derived  
32 type definition must contain the PRIVATE statement or the derived type must be pri-  
33 vate.
- 34 Constraint: A derived type *type-name* must not be the same as the name of any intrinsic type nor  
35 the same as any other accessible derived *type-name*.
- 36 R425 *end-type-stmt* is END TYPE [ *type-name* ]
- 37 Constraint: If END TYPE is followed by a *type-name*, the *type-name* must be the same as that in the  
38 corresponding *derived-type-stmt*.
- 39 R426 *component-def-stmt* is *type-spec* [ [ , *component-attr-spec-list* ] :: ] ■  
40 ■ *component-decl-list*
- 41 R427 *component-attr-spec* is POINTER  
42 or DIMENSION ( *component-array-spec* )
- 43 Constraint: No *component-attr-spec* may appear more than once in a given *component-def-stmt*.
- 44 Constraint: If the POINTER attribute is not specified for a component, a *type-spec* in the  
45 *component-def-stmt* must specify an intrinsic type or a previously defined derived  
46 type.

- 1 Constraint: If the POINTER attribute is specified for a component, a *type-spec* in the *component-*  
 2 *def-stmt* must specify an intrinsic type or any accessible derived type including the  
 3 type being defined.
- 4 R428 *component-array-spec* is *explicit-shape-spec-list*  
 5 or *deferred-shape-spec-list*
- 6 R429 *component-decl* is *component-name* [ ( *component-array-spec* ) ] ■  
 7 ■ [ \* *char-length* ]
- 8 Constraint: If the POINTER attribute is not specified, each *component-array-spec* must be an  
 9 *explicit-shape-spec-list*.
- 10 Constraint: If the POINTER attribute is specified, each *component-array-spec* must be a *deferred-*  
 11 *shape-spec-list*.
- 12 Constraint: The \* *char-length* option is permitted only if the type specified is character.
- 13 Constraint: A *char-length* in a *component-decl* must be a constant specification expression (7.1.6.2).
- 14 Constraint: Each bound in the *explicit-shape-spec* (R428) must be a constant specification expres-  
 15 sion (7.1.6.2).
- 16 R430 *structure-constructor* is *type-name* ( *expr-list* )
- 17 R431 *array-constructor* is ( / *ac-value-list* / )
- 18 R432 *ac-value* is *expr*  
 19 or *ac-implied-do*
- 20 R433 *ac-implied-do* is ( *ac-value-list* , *ac-implied-do-control* )
- 21 R434 *ac-implied-do-control* is *ac-do-variable* = *scalar-int-expr* , ■  
 22 ■ *scalar-int-expr* [ , *scalar-int-expr* ]
- 23 R435 *ac-do-variable* is *scalar-int-variable*
- 24 Constraint: *ac-do-variable* must be a named variable.
- 25 Constraint: Each *ac-value* expression in the sequence must have the same type and type paramete-  
 26 rs.
- 27 **D.1.5 Data Object Declarations and Specifications.**
- 28 R501 *type-declaration-stmt* is *type-spec* [ [ , *attr-spec* ] ... :: ] *entity-decl-list*
- 29 R502 *type-spec* is INTEGER [ *kind-selector* ]  
 30 or REAL [ *kind-selector* ]  
 31 or DOUBLE PRECISION  
 32 or COMPLEX [ *kind-selector* ]  
 33 or CHARACTER [ *char-selector* ]  
 34 or LOGICAL [ *kind-selector* ]  
 35 or TYPE ( *type-name* )
- 36 R503 *attr-spec* is PARAMETER  
 37 or *access-spec*  
 38 or ALLOCATABLE  
 39 or DIMENSION ( *array-spec* )  
 40 or EXTERNAL  
 41 or INTENT ( *intent-spec* )  
 42 or INTRINSIC  
 43 or OPTIONAL  
 44 or POINTER  
 45 or SAVE  
 46 or TARGET

## SYNTAX RULES

- 1 R504 *entity-decl* is *object-name* [ ( *array-spec* ) ] ■  
 2 ■ [ \* *char-length* ] [= *initialization-expr* ]  
 3 or *function-name* [ ( *array-spec* ) ] [ \* *char-length* ]
- 4 R505 *kind-selector* is ( [ KIND = ] *scalar-int-initialization-expr* )
- 5 Constraint: The same *attr-spec* must not appear more than once in a given *type-declaration-stmt*.
- 6 Constraint: The *function-name* must be the name of an external function, an intrinsic function, a  
 7 function dummy procedure, or a statement function.
- 8 Constraint: The = *initialization-expr* must appear if the statement contains a PARAMETER attri-  
 9 bute (5.1.2.1).
- 10 Constraint: If = *initialization-expr* appears, a double colon separator must appear before the  
 11 *entity-decl-list*.
- 12 Constraint: The = *initialization-expr* must not appear if *object-name* is a dummy argument, a func-  
 13 tion result, an object in a named common block unless the type declaration is in a  
 14 block data program unit, an object in blank common, an allocatable array, a pointer,  
 15 an external name, an intrinsic name, or an automatic object.
- 16 Constraint: The \* *char-length* option is permitted only if the type specified is character.
- 17 Constraint: The ALLOCATABLE attribute may be used only when declaring an array that is not  
 18 a dummy argument or a function result.
- 19 Constraint: An array declared with a POINTER or an ALLOCATABLE attribute must be speci-  
 20 fied with an *array-spec* that is a *deferred-shape-spec-list* (5.1.2.4.3).
- 21 Constraint: An *array-spec* for a *function-name* that does not have the pointer attribute must be an  
 22 *explicit-shape-spec-list*.
- 23 Constraint: An *array-spec* for a *function-name* that does have the pointer attribute must be a  
 24 *deferred-shape-spec-list*.
- 25 Constraint: If the POINTER attribute is specified, the INTENT, EXTERNAL, or INTRINSIC attri-  
 26 bute must not be specified.
- 27 Constraint: If the TARGET attribute is specified, the EXTERNAL, INTRINSIC, or PARAMETER  
 28 attribute must not be specified.
- 29 Constraint: The PARAMETER attribute must not be specified for dummy arguments, pointers,  
 30 functions, or objects in a common block.
- 31 Constraint: The INTENT and OPTIONAL attributes may be specified only for dummy argu-  
 32 ments.
- 33 Constraint: An entity must not have the PUBLIC attribute if its type has the PRIVATE attribute.
- 34 Constraint: The SAVE attribute must not be specified for an object that is in a common block, a  
 35 dummy argument, a procedure, a function result, or an automatic data object.
- 36 Constraint: ~~An~~ entity must not have the EXTERNAL attribute if it has the INTRINSIC attribute.
- 37 Constraint: ~~An~~ entity in a *type-declaration-stmt* must not have the EXTERNAL or INTRINSIC  
 38 attribute specified unless it is a function.
- 39 Constraint: An array must not have both the ALLOCATABLE attribute and the POINTER attri-  
 40 bute.
- 41 Constraint: An entity must not be given explicitly any attribute more than once in a scoping unit.
- 42 Constraint: The value specified in a *kind-selector* must be nonnegative.
- 43 R506 *char-selector* is *length-selector*  
 44 or ( [ LEN = ] *type-param-value* , ■  
 45 ■ [ KIND = ] *scalar-int-initialization-expr* )  
 46 or ( KIND = *scalar-int-initialization-expr* ■



## SYNTAX RULES

- 1 R521 *access-stmt* is *access-spec* [ [ :: ] *access-id-list* ]
- 2 R522 *access-id* is *use-name*
- 3 or *generic-spec*
- 4 Constraint: An *access-stmt* may appear only in the scoping unit of a module. Only one accessibility  
5 statement with an omitted *access-id-list* is permitted in the scoping unit of a mod-  
6 ule.
- 7 Constraint: Each *use-name* must be the name of a named variable, nonintrinsic procedure, generic  
8 procedure, derived type, named constant, or namelist group.
- 9 Constraint: A module procedure that has a dummy argument or function result of a type that  
10 has PRIVATE accessibility must have PRIVATE accessibility and must not have a  
11 generic identifier that has PUBLIC accessibility.
- 12 R523 *save-stmt* is SAVE [ [ :: ] *saved-entity-list* ]
- 13 R524 *saved-entity* is *object-name*
- 14 or / *common-block-name* /
- 15 Constraint: An *object-name* must not be a dummy argument name, a procedure name, a function  
16 result name, an automatic data object name, a namelist group name, or the name of  
17 an entity in a common block.
- 18 Constraint: If a SAVE statement with an omitted saved entity list occurs in a scoping unit, no  
19 other explicit occurrence of the SAVE attribute or SAVE statement is permitted in the  
20 same scoping unit.
- 21 R525 *dimension-stmt* is DIMENSION [ [ :: ] *array-name* ( *array-spec* ) ■  
22 ■ [ , *array-name* ( *array-spec* ) ] ...
- 23 R526 *allocatable-stmt* is ALLOCATABLE [ [ :: ] *array-name* ■  
24 ■ [ ( *deferred-shape-spec-list* ) ] ■  
25 ■ [ , *array-name* [ ( *deferred-shape-spec-list* ) ] ] ...
- 26 Constraint: The *array-name* must not be a dummy argument or function result.
- 27 Constraint: If the DIMENSION attribute for an *array-name* is specified elsewhere in the scoping  
28 unit, the *array-spec* must be a *deferred-shape-spec-list*.
- 29 R527 *pointer-stmt* is POINTER [ [ :: ] *object-name* ■  
30 ■ [ ( *deferred-shape-spec-list* ) ] ■  
31 ■ [ , *object-name* [ ( *deferred-shape-spec-list* ) ] ] ...
- 32 Constraint: The INTENT attribute must not be specified for an *object-name*.
- 33 Constraint: If the DIMENSION attribute for an *object-name* is specified elsewhere in the scoping  
34 unit, the *array-spec* must be a *deferred-shape-spec-list*.
- 35 Constraint: The PARAMETER attribute must not be specified for *object-name*.
- 36 R528 *target-stmt* is TARGET [ [ :: ] *object-name* [ ( *array-spec* ) ] ■  
37 ■ [ , *object-name* [ ( *array-spec* ) ] ] ...
- 38 Constraint: The PARAMETER attribute must not be specified for an *object-name*.
- 39 R529 *data-stmt* is DATA *data-stmt-set* [ [ , ] *data-stmt-set* ] ...
- 40 R530 *data-stmt-set* is *data-stmt-object-list* / *data-stmt-value-list* /
- 41 R531 *data-stmt-object* is *variable*
- 42 or *data-implied-do*
- 43 R532 *data-stmt-value* is [ *data-stmt-repeat* \* ] *data-stmt-constant*
- 44 R533 *data-stmt-constant* is *scalar-constant*
- 45 or *signed-int-literal-constant*
- 46 or *signed-real-literal-constant*



- 1 or *structure-constructor*  
 2 or *boz-literal-constant*
- 3 R534 *data-stmt-repeat* is *scalar-int-constant*
- 4 R535 *data-implied-do* is ( *data-i-do-object-list* , *data-i-do-variable* = ■  
 5 ■ *scalar-int-expr* , *scalar-int-expr* [ , *scalar-int-expr* ] )
- 6 R536 *data-i-do-object* is *array-element*  
 7 or *data-implied-do*
- 8 Constraint: *array-element* must not be a subobject with a constant parent.
- 9 R537 *data-i-do-variable* is *scalar-int-variable*
- 10 Constraint: *data-i-do-variable* must be a named variable.
- 11 Constraint: The DATA statement repeat factor must be positive or zero. If the DATA statement  
 12 repeat factor is a named constant, it must have been declared previously in the scop-  
 13 ing unit or made accessible by use association or host association.
- 14 Constraint: If a *data-stmt-constant* is a *structure-constructor*, each component must be a constant  
 15 expression.
- 16 Constraint: A variable whose name or designator is included in a *data-stmt-object-list* or a *data-i-*  
 17 *do-object-list* must not be: a dummy argument, made accessible by use association or  
 18 host association, in a named common block unless the DATA statement is in a block  
 19 data program unit, in a blank common block, a function name, a function result  
 20 name, an automatic object, a pointer, or an allocatable array.
- 21 Constraint: A subscript in an array element *data-i-do-object* must be an expression whose prima-  
 22 ries are either constants or DO variables of the containing *data-implied-dos*.
- 23 Constraint: A *scalar-int-expr* of a *data-implied-do* must involve as primaries only constants or DO  
 24 variables of the containing *data-implied-dos*.
- 25 R538 *parameter-stmt* is PARAMETER ( *named-constant-def-list* )
- 26 R539 *named-constant-def* is *named-constant* = *initialization-expr*
- 27 R540 *implicit-stmt* is IMPLICIT *implicit-spec-list*  
 28 or IMPLICIT NONE
- 29 R541 *implicit-spec* is *type-spec* ( *letter-spec-list* )
- 30 R542 *letter-spec* is *letter* [ - *letter* ]
- 31 Constraint: If IMPLICIT NONE is specified in a scoping unit, it must precede any PARAMETER  
 32 statements that appear in the scoping unit and there must be no other IMPLICIT  
 33 statements in the scoping unit.
- 34 Constraint: If the minus and second letter appear, the second letter must follow the first letter  
 35 alphabetically.
- 36 R543 *namelist-stmt* is NAMELIST / *namelist-group-name* / ■  
 37 ■ *namelist-group-object-list* ■  
 38 ■ [ [ , ] / *namelist-group-name* / ■  
 39 ■ *namelist-group-object-list* ] ...
- 40 R544 *namelist-group-object* is *variable-name*
- 41 Constraint: A *namelist-group-object* must not be an array dummy argument with nonconstant  
 42 bounds, a variable with assumed character length, an automatic object, a pointer, of a  
 43 type that has an ultimate component that is a pointer, an allocatable array, or a sub-  
 44 object of any of the preceding objects.
- 45 Constraint: If a *namelist-group-name* has the PUBLIC attribute, no item in the *namelist-group-*  
 46 *object-list* may have the PRIVATE attribute.

## SYNTAX RULES

- 1 R545 *equivalence-stmt* is EQUIVALENCE *equivalence-set-list*  
2 R546 *equivalence-set* is ( *equivalence-object* , *equivalence-object-list* )  
3 R547 *equivalence-object* is *variable-name*  
4 or *array-element*  
5 or *substring*
- 6 Constraint: An *equivalence-object* must not be a dummy argument, a pointer, an allocatable array,  
7 an object of a nonsequence derived type or of a sequence derived type containing a  
8 pointer, an automatic object, a function name, an entry name, a result name, a named  
9 constant, or a subobject of any of the preceding objects.
- 10 Constraint: Each subscript or substring range expression in an *equivalence-object* must be an inte-  
11 ger initialization expression (7.1.6.1).
- 12 Constraint: If an *equivalence-object* is of type default integer, default real, double precision real,  
13 default complex, default logical, or numeric sequence type, all of the objects in the  
14 equivalence set must be of these types.
- 15 Constraint: If an *equivalence-object* is of type default character or character sequence type, all of  
16 the objects in the equivalence set must be of these types.
- 17 Constraint: If an *equivalence-object* is of a derived type that is not a numeric sequence or character  
18 sequence type, all of the objects in the equivalence set must be of the same type.
- 19 Constraint: If an *equivalence-object* is of an intrinsic type other than default integer, default real,  
20 double precision real, default complex, default logical, or default character, all of the  
21 objects in the equivalence set must be of the same type with the same kind type  
22 parameter value.
- 23 R548 *common-stmt* is COMMON [ / [ *common-block-name* ] / ] ■  
24 ■ *common-block-object-list* ■  
25 ■ [ [ , ] / [ *common-block-name* ] / ■  
26 ■ *common-block-object-list* ] ...
- 27 R549 *common-block-object* is *variable-name* [ ( *explicit-shape-spec-list* ) ]
- 28 Constraint: Only one appearance of a given *variable-name* is permitted in all *common-block-object-*  
29 *lists* within a scoping unit.
- 30 Constraint: A *common-block-object* must not be a dummy argument, an allocatable array, an auto-  
31 matic object, a function name, an entry name, or a result name.
- 32 Constraint: Each bound in the *explicit-shape-spec* must be a constant specification expression  
33 (7.1.6.2).
- 34 Constraint: If a *common-block-object* is of a derived type, it must be a sequence type (4.4.1).
- 35 Constraint: If a *variable-name* appears with an *explicit-shape-spec-list*, it must not have the  
36 POINTER attribute.
- 37 **D.1.6 Use of Data Objects.**
- 38 R601 *variable* is *scalar-variable-name*  
39 or *array-variable-name*  
40 or *subobject*
- 41 Constraint: *array-variable-name* must be the name of a data object that is an array.
- 42 Constraint: *array-variable-name* must not have the PARAMETER attribute.
- 43 Constraint: *scalar-variable-name* must not have the PARAMETER attribute.
- 44 Constraint: *subobject* must not be a subobject designator (for example, a substring) whose parent  
45 is a constant.

- 1 R602 *subobject* is *array-element*  
 2 or *array-section*  
 3 or *structure-component*  
 4 or *substring*
- 5 R603 *logical-variable* is *variable*
- 6 Constraint: *logical-variable* must be of type logical.
- 7 R604 *default-logical-variable* is *variable*
- 8 Constraint: *default-logical-variable* must be of type default logical.
- 9 R605 *char-variable* is *variable*
- 10 Constraint: *char-variable* must be of type character.
- 11 R606 *default-char-variable* is *variable*
- 12 Constraint: *default-char-variable* must be of type default character.
- 13 R607 *int-variable* is *variable*
- 14 Constraint: *int-variable* must be of type integer.
- 15 R608 *default-int-variable* is *variable*
- 16 Constraint: *default-int-variable* must be of type default integer.
- 17 R609 *substring* is *parent-string* ( *substring-range* )
- 18 R610 *parent-string* is *scalar-variable-name*  
 19 or *array-element*  
 20 or *scalar-structure-component*  
 21 or *scalar-constant*
- 22 R611 *substring-range* is [ *scalar-int-expr* ] : [ *scalar-int-expr* ]
- 23 Constraint: *parent-string* must be of type character.
- 24 R612 *data-ref* is *part-ref* [ % *part-ref* ] ...
- 25 R613 *part-ref* is *part-name* [ ( *section-subscript-list* ) ]
- 26 Constraint: In a *data-ref*, each *part-name* except the rightmost must be of derived type.
- 27 Constraint: In a *data-ref*, each *part-name* except the leftmost must be the name of a component of  
 28 the derived type definition of the type of the preceding *part-name*.
- 29 Constraint: In a *part-ref* containing a *section-subscript-list*, the number of *section-subscripts* must  
 30 equal the rank of *part-name*.
- 31 Constraint: In a *data-ref*, there must not be more than one *part-ref* with nonzero rank.
- 32 R614 *structure-component* is *data-ref*
- 33 Constraint: In a *structure-component*, there must be more than one *part-ref* and the rightmost *part-*  
 34 *ref* must be of the form *part-name*.
- 35 R615 *array-element* is *data-ref*
- 36 Constraint: In an *array-element*, every *part-ref* must have rank zero and at least one *part-ref* must  
 37 contain a *subscript-list*.
- 38 R616 *array-section* is *data-ref* [ ( *substring-range* ) ]
- 39 Constraint: In an *array-section*, one *part-ref* must have a *section-subscript-list* and have nonzero  
 40 rank.
- 41 Constraint: In an *array-section* with a *substring-range*, the rightmost *part-name* must be of type  
 42 character.

## SYNTAX RULES

- 1 R617 *subscript* is *scalar-int-expr*
- 2 R618 *section-subscript* is *subscript*
- 3 or *subscript-triplet*
- 4 or *vector-subscript*
- 5 R619 *subscript-triplet* is [ *subscript* ] : [ *subscript* ] [ : *stride* ]
- 6 R620 *stride* is *scalar-int-expr*
- 7 R621 *vector-subscript* is *int-expr*
- 8 Constraint: A *vector-subscript* must be an integer array expression of rank one.
- 9 Constraint: The second *subscript* must not be omitted from a *subscript-triplet* in the last dimension
- 10 of an assumed-size array.
- 11 R622 *allocate-stmt* is ALLOCATE ( *allocation-list* ■
- 12 ■ [ , STAT = *stat-variable* ] )
- 13 R623 *stat-variable* is *scalar-int-variable*
- 14 Constraint: The *stat-variable* must not be allocated within the ALLOCATE statement in which it
- 15 appears.
- 16 R624 *allocation* is *allocate-object* [ ( *explicit-shape-spec-list* ) ]
- 17 R625 *allocate-object* is *variable-name*
- 18 or *structure-component*
- 19 Constraint: Each *allocate-object* must be a pointer or an allocatable array.
- 20 Constraint: A bound in an *allocation explicit-shape-spec* must not be an expression involving as a
- 21 primary an array inquiry function (13.10.15) whose argument is any other object in
- 22 the same ALLOCATE statement.
- 23 Constraint: The number of *explicit-shape-specs* in an *allocation explicit-shape-spec-list* must be the
- 24 same as the rank of the pointer or allocatable array.
- 25 R626 *nullify-stmt* is NULLIFY ( *pointer-object-list* )
- 26 R627 *pointer-object* is *variable-name*
- 27 or *structure-component*
- 28 Constraint: Each *pointer-object* must have the POINTER attribute.
- 29 R628 *deallocate-stmt* is DEALLOCATE ( *allocate-object-list* ■
- 30 ■ [ , STAT = *stat-variable* ] )
- 31 Constraint: Each *allocate-object* must be a pointer or an allocatable array.
- 32 Constraint: The *stat-variable* must not be deallocated within the same DEALLOCATE statement.

### 33 D.1.7 Expressions and Assignment.

- 34 R701 *primary* is *constant*
- 35 or *constant-subobject*
- 36 or *variable*
- 37 or *array-constructor*
- 38 or *structure-constructor*
- 39 or *function-reference*
- 40 or ( *expr* )
- 41 R702 *constant-subobject* is *subobject*
- 42 Constraint: *subobject* must be a subobject designator whose parent is a constant.
- 43 Constraint: A *variable* that is a *primary* must not be an assumed-size array.

|    |                                                                                                          |                          |                                                                 |
|----|----------------------------------------------------------------------------------------------------------|--------------------------|-----------------------------------------------------------------|
| 1  | R703                                                                                                     | <i>level-1-expr</i>      | is [ <i>defined-unary-op</i> ] <i>primary</i>                   |
| 2  | R704                                                                                                     | <i>defined-unary-op</i>  | is . <i>letter</i> [ <i>letter</i> ] ... .                      |
| 3  | Constraint: A <i>defined-unary-op</i> must not contain more than 31 letters and must not be the same as  |                          |                                                                 |
| 4  | any <i>intrinsic-operator</i> or <i>logical-literal-constant</i> .                                       |                          |                                                                 |
| 5  | R705                                                                                                     | <i>mult-operand</i>      | is <i>level-1-expr</i> [ <i>power-op mult-operand</i> ]         |
| 6  | R706                                                                                                     | <i>add-operand</i>       | is [ <i>add-operand mult-op</i> ] <i>mult-operand</i>           |
| 7  | R707                                                                                                     | <i>level-2-expr</i>      | is [ [ <i>level-2-expr</i> ] <i>add-op</i> ] <i>add-operand</i> |
| 8  | R708                                                                                                     | <i>power-op</i>          | is **                                                           |
| 9  | R709                                                                                                     | <i>mult-op</i>           | is *                                                            |
| 10 |                                                                                                          |                          | or /                                                            |
| 11 | R710                                                                                                     | <i>add-op</i>            | is +                                                            |
| 12 |                                                                                                          |                          | or -                                                            |
| 13 | R711                                                                                                     | <i>level-3-expr</i>      | is [ <i>level-3-expr concat-op</i> ] <i>level-2-expr</i>        |
| 14 | R712                                                                                                     | <i>concat-op</i>         | is //                                                           |
| 15 | R713                                                                                                     | <i>level-4-expr</i>      | is [ <i>level-3-expr rel-op</i> ] <i>level-3-expr</i>           |
| 16 | R714                                                                                                     | <i>rel-op</i>            | is .EQ.                                                         |
| 17 |                                                                                                          |                          | or .NE.                                                         |
| 18 |                                                                                                          |                          | or .LT.                                                         |
| 19 |                                                                                                          |                          | or .LE.                                                         |
| 20 |                                                                                                          |                          | or .GT.                                                         |
| 21 |                                                                                                          |                          | or .GE.                                                         |
| 22 |                                                                                                          |                          | or ==                                                           |
| 23 |                                                                                                          |                          | or /=                                                           |
| 24 |                                                                                                          |                          | or <                                                            |
| 25 |                                                                                                          |                          | or <=                                                           |
| 26 |                                                                                                          |                          | or >                                                            |
| 27 |                                                                                                          |                          | or >=                                                           |
| 28 | R715                                                                                                     | <i>and-operand</i>       | is [ <i>not-op</i> ] <i>level-4-expr</i>                        |
| 29 | R716                                                                                                     | <i>or-operand</i>        | is [ <i>or-operand and-op</i> ] <i>and-operand</i>              |
| 30 | R717                                                                                                     | <i>equiv-operand</i>     | is [ <i>equiv-operand or-op</i> ] <i>or-operand</i>             |
| 31 | R718                                                                                                     | <i>level-5-expr</i>      | is [ <i>level-5-expr equiv-op</i> ] <i>equiv-operand</i>        |
| 32 | R719                                                                                                     | <i>not-op</i>            | is .NOT.                                                        |
| 33 | R720                                                                                                     | <i>and-op</i>            | is .AND.                                                        |
| 34 | R721                                                                                                     | <i>or-op</i>             | is .OR.                                                         |
| 35 | R722                                                                                                     | <u><i>equiv-op</i></u>   | is .EQV.                                                        |
| 36 |                                                                                                          |                          | or .NEQV.                                                       |
| 37 | R723                                                                                                     | <u><i>expr</i></u>       | is [ <i>expr defined-binary-op</i> ] <i>level-5-expr</i>        |
| 38 | R724                                                                                                     | <i>defined-binary-op</i> | is . <i>letter</i> [ <i>letter</i> ] ... .                      |
| 39 | Constraint: A <i>defined-binary-op</i> must not contain more than 31 letters and must not be the same as |                          |                                                                 |
| 40 | any <i>intrinsic-operator</i> or <i>logical-literal-constant</i> .                                       |                          |                                                                 |
| 41 | R725                                                                                                     | <i>logical-expr</i>      | is <i>expr</i>                                                  |
| 42 | Constraint: <i>logical-expr</i> must be type logical.                                                    |                          |                                                                 |
| 43 | R726                                                                                                     | <i>char-expr</i>         | is <i>expr</i>                                                  |

## SYNTAX RULES

- 1 Constraint: *char-expr* must be type character.
- 2 R727 *default-char-expr* is *expr*
- 3 Constraint: *default-char-expr* must be of type default character.
- 4 R728 *int-expr* is *expr*
- 5 Constraint: *int-expr* must be type integer.
- 6 R729 *numeric-expr* is *expr*
- 7 Constraint: *numeric-expr* must be of type integer, real or complex.
- 8 R730 *initialization-expr* is *expr*
- 9 R731 *char-initialization-expr* is *char-expr*
- 10 R732 *int-initialization-expr* is *int-expr*
- 11 R733 *logical-initialization-expr* is *logical-expr*
- 12 R734 *specification-expr* is *scalar-int-expr*
- 13 Constraint: The *scalar-int-expr* must be a restricted expression.
- 14 R735 *assignment-stmt* is *variable = expr*
- 15 Constraint: A *variable* in an *assignment-stmt* must not be an assumed-size array.
- 16 R736 *pointer-assignment-stmt* is *pointer-object => target*
- 17 R737 *target* is *variable*
- 18 or *expr*
- 19 Constraint: The *pointer-object* must have the POINTER attribute.
- 20 Constraint: The *variable* must have the TARGET attribute or be a subobject of an object with the
- 21 TARGET attribute, or it must have the POINTER attribute or be a subobject, other
- 22 than a structure component of an object with the POINTER attribute.
- 23 Constraint: The *target* must be of the same type, type parameters, and rank as the pointer.
- 24 Constraint: The *target* must not be an array section with a vector subscript.
- 25 Constraint: The *expr* must deliver a pointer result.
- 26 R738 *where-stmt* is WHERE ( *mask-expr* ) *assignment-stmt*
- 27 R739 *where-construct* is *where-construct-stmt*
- 28 [ *assignment-stmt* ] ...
- 29 [ *elsewhere-stmt*
- 30 [ *assignment-stmt* ] ... ]
- 31 *end-where-stmt*
- 32 R740 *where-construct-stmt* is WHERE ( *mask-expr* )
- 33 R741 *mask-expr* is *logical-expr*
- 34 R742 *elsewhere-stmt* is ELSEWHERE
- 35 R743 *end-where-stmt* is END WHERE
- 36 Constraint: In each *assignment-stmt*, the *mask-expr* and the variable being defined must be arrays
- 37 of the same shape.
- 38 Constraint: The *assignment-stmt* must not be a defined assignment.
- 39 **D.1.8 Execution Control.**
- 40 R801 *block* is [ *execution-part-construct* ] ...

|    |                                                                                                                                         |                         |                                                                             |
|----|-----------------------------------------------------------------------------------------------------------------------------------------|-------------------------|-----------------------------------------------------------------------------|
| 1  | R802                                                                                                                                    | <i>if-construct</i>     | is <i>if-then-stmt</i>                                                      |
| 2  |                                                                                                                                         |                         | <i>block</i>                                                                |
| 3  |                                                                                                                                         |                         | [ <i>else-if-stmt</i>                                                       |
| 4  |                                                                                                                                         |                         | <i>block</i> ] ...                                                          |
| 5  |                                                                                                                                         |                         | [ <i>else-stmt</i>                                                          |
| 6  |                                                                                                                                         |                         | <i>block</i> ]                                                              |
| 7  |                                                                                                                                         |                         | <i>end-if-stmt</i>                                                          |
| 8  | R803                                                                                                                                    | <i>if-then-stmt</i>     | is [ <i>if-construct-name</i> : ] IF ( <i>scalar-logical-expr</i> ) THEN    |
| 9  | R804                                                                                                                                    | <i>else-if-stmt</i>     | is ELSE IF ( <i>scalar-logical-expr</i> ) THEN [ <i>if-construct-name</i> ] |
| 10 | R805                                                                                                                                    | <i>else-stmt</i>        | is ELSE [ <i>if-construct-name</i> ]                                        |
| 11 | R806                                                                                                                                    | <i>end-if-stmt</i>      | is END IF [ <i>if-construct-name</i> ]                                      |
| 12 | Constraint: If the <i>if-then-stmt</i> of an <i>if-construct</i> is identified by an <i>if-construct-name</i> , the corre-              |                         |                                                                             |
| 13 | sponding <i>end-if-stmt</i> must specify the same <i>if-construct-name</i> . If the <i>if-then-stmt</i> of an                           |                         |                                                                             |
| 14 | <i>if-construct</i> is not identified by an <i>if-construct-name</i> , the corresponding <i>end-if-stmt</i>                             |                         |                                                                             |
| 15 | must not specify an <i>if-construct-name</i> . If an <i>else-if-stmt</i> or <i>else-stmt</i> is identified by an                        |                         |                                                                             |
| 16 | <i>if-construct-name</i> , the corresponding <i>if-then-stmt</i> must specify the same <i>if-construct-</i>                             |                         |                                                                             |
| 17 | <i>name</i> .                                                                                                                           |                         |                                                                             |
| 18 | R807                                                                                                                                    | <i>if-stmt</i>          | is IF ( <i>scalar-logical-expr</i> ) <i>action-stmt</i>                     |
| 19 | Constraint: The <i>action-stmt</i> in the <i>if-stmt</i> must not be an <i>if-stmt</i> , <i>end-program-stmt</i> , <i>end-function-</i> |                         |                                                                             |
| 20 | <i>stmt</i> , or <i>end-subroutine-stmt</i> .                                                                                           |                         |                                                                             |
| 21 | R808                                                                                                                                    | <i>case-construct</i>   | is <i>select-case-stmt</i>                                                  |
| 22 |                                                                                                                                         |                         | [ <i>case-stmt</i>                                                          |
| 23 |                                                                                                                                         |                         | <i>block</i> ] ...                                                          |
| 24 |                                                                                                                                         |                         | <i>end-select-stmt</i>                                                      |
| 25 | R809                                                                                                                                    | <i>select-case-stmt</i> | is [ <i>case-construct-name</i> : ] SELECT CASE ( <i>case-expr</i> )        |
| 26 | R810                                                                                                                                    | <i>case-stmt</i>        | is CASE <i>case-selector</i> [ <i>case-construct-name</i> ]                 |
| 27 | R811                                                                                                                                    | <i>end-select-stmt</i>  | is END SELECT [ <i>case-construct-name</i> ]                                |
| 28 | Constraint: If the <i>select-case-stmt</i> of a <i>case-construct</i> is identified by a <i>case-construct-name</i> , the corre-        |                         |                                                                             |
| 29 | sponding <i>end-select-stmt</i> must specify the same <i>case-construct-name</i> . If the <i>select-case-</i>                           |                         |                                                                             |
| 30 | <i>stmt</i> of a <i>case-construct</i> is not identified by a <i>case-construct-name</i> , the corresponding                            |                         |                                                                             |
| 31 | <i>end-select-stmt</i> must not specify a <i>case-construct-name</i> . If a <i>case-stmt</i> is identified by a                         |                         |                                                                             |
| 32 | <i>case-construct-name</i> , the corresponding <i>select-case-stmt</i> must specify the same <i>case-</i>                               |                         |                                                                             |
| 33 | <i>construct-name</i> .                                                                                                                 |                         |                                                                             |
| 34 | R812                                                                                                                                    | <i>case-expr</i>        | is <i>scalar-int-expr</i>                                                   |
| 35 |                                                                                                                                         |                         | or <i>scalar-char-expr</i>                                                  |
| 36 |                                                                                                                                         |                         | or <i>scalar-logical-expr</i>                                               |
| 37 | R813                                                                                                                                    | <i>case-selector</i>    | is ( <i>case-value-range-list</i> )                                         |
| 38 |                                                                                                                                         |                         | or DEFAULT                                                                  |
| 39 | Constraint: <u>No</u> more than one of the selectors of one of the CASE statements may be DEFAULT.                                      |                         |                                                                             |
| 40 | R814                                                                                                                                    | <i>case-value-range</i> | is <i>case-value</i>                                                        |
| 41 |                                                                                                                                         |                         | or <i>case-value</i> :                                                      |
| 42 |                                                                                                                                         |                         | or : <i>case-value</i>                                                      |
| 43 |                                                                                                                                         |                         | or <i>case-value</i> : <i>case-value</i>                                    |
| 44 | R815                                                                                                                                    | <i>case-value</i>       | is <i>scalar-int-initialization-expr</i>                                    |
| 45 |                                                                                                                                         |                         | or <i>scalar-char-initialization-expr</i>                                   |
| 46 |                                                                                                                                         |                         | or <i>scalar-logical-initialization-expr</i>                                |
| 47 | Constraint: For a given <i>case-construct</i> , each <i>case-value</i> must be of the same type as <i>case-expr</i> . For               |                         |                                                                             |
| 48 | character type, length differences are allowed, but the kind type parameters must be                                                    |                         |                                                                             |

## SYNTAX RULES

- 1 the same.
- 2 Constraint: A *case-value-range* using a colon must not be used if *case-expr* is of type logical.
- 3 Constraint: For a given *case-construct*, the *case-value-ranges* must not overlap; that is, there must  
4 be no possible value of the *case-expr* that matches more than one *case-value-range*.
- 5 R816 *do-construct* is *block-do-construct*  
6 or *nonblock-do-construct*
- 7 R817 *block-do-construct* is *do-stmt*  
8 *do-block*  
9 *end-do*
- 10 R818 *do-stmt* is *label-do-stmt*  
11 or *nonlabel-do-stmt*
- 12 R819 *label-do-stmt* is [ *do-construct-name* : ] DO *label* [ *loop-control* ]
- 13 R820 *nonlabel-do-stmt* is [ *do-construct-name* : ] DO [ *loop-control* ]
- 14 R821 *loop-control* is [ , ] *do-variable* = *scalar-numeric-expr* , ■  
15 ■ *scalar-numeric-expr* [ , *scalar-numeric-expr* ]  
16 or [ , ] WHILE ( *scalar-logical-expr* )
- 17 R822 *do-variable* is *scalar-variable*
- 18 Constraint: The *do-variable* must be a named scalar variable of type integer, default real, or double pre-  
19 cision real.
- 20 Constraint: Each *scalar-numeric-expr* in *loop-control* must be of type integer, default real, or double pre-  
21 cision real.
- 22 R823 *do-block* is *block*
- 23 R824 *end-do* is *end-do-stmt*  
24 or *continue-stmt*
- 25 R825 *end-do-stmt* is END DO [ *do-construct-name* ]
- 26 Constraint: If the *do-stmt* of a *block-do-construct* is identified by a *do-construct-name*, the corre-  
27 sponding *end-do* must be an *end-do-stmt* specifying the same *do-construct-name*. If the  
28 *do-stmt* of a *block-do-construct* does not so specify a *do-construct-name*, the correspond-  
29 ing *end-do* must not specify a *do-construct-name*.
- 30 Constraint: If the *do-stmt* is a *nonlabel-do-stmt*, the corresponding *end-do* must be an *end-do-stmt*.
- 31 Constraint: If the *do-stmt* is a *label-do-stmt*, the corresponding *end-do* must be identified with the  
32 same *label*.
- 33 R826 *nonblock-do-construct* is *action-term-do-construct*  
34 or *outer-shared-do-construct*
- 35 R827 *action-term-do-construct* is *label-do-stmt*  
36 *do-body*  
37 *do-term-action-stmt*
- 38 R828 *do-body* is [ *execution-part-construct* ] ...
- 39 R829 *do-term-action-stmt* is *action-stmt*
- 40 Constraint: A *do-term-action-stmt* must not be a *continue-stmt*, a *goto-stmt*, a *return-stmt*, a *stop-stmt*, an *exit-stmt*, a *cycle-*  
41 *stmt*, an *end-function-stmt*, an *end-subroutine-stmt*, an *end-program-stmt*, an *arithmetic-if-stmt*, or an *assigned-*  
42 *goto-stmt*.
- 43 Constraint: The *do-term-action-stmt* must be identified with a label and the corresponding *label-do-stmt* must refer to the  
44 same label.
- 45 R830 *outer-shared-do-construct* is *label-do-stmt*





## SYNTAX RULES

### 1 D.1.9 Input/Output Statements.

- 2 R901 *io-unit* is *external-file-unit*  
 3 or \*  
 4 or *internal-file-unit*
- 5 R902 *external-file-unit* is *scalar-int-expr*
- 6 R903 *internal-file-unit* is *default-char-variable*
- 7 Constraint: The *char-variable* must not be an array section with a vector subscript.
- 8 R904 *open-stmt* is OPEN ( *connect-spec-list* )
- 9 R905 *connect-spec* is [ UNIT= ] *external-file-unit*  
 10 or IOSTAT= *scalar-default-int-variable*  
 11 or ERR= *label*  
 12 or FILE= *file-name-expr*  
 13 or STATUS= *scalar-default-char-expr*  
 14 or ACCESS= *scalar-default-char-expr*  
 15 or FORM= *scalar-default-char-expr*  
 16 or RECL= *scalar-int-expr*  
 17 or BLANK= *scalar-default-char-expr*  
 18 or POSITION= *scalar-default-char-expr*  
 19 or ACTION= *scalar-default-char-expr*  
 20 or DELIM= *scalar-default-char-expr*  
 21 or PAD= *scalar-default-char-expr*
- 22 R906 *file-name-expr* is *scalar-default-char-expr*
- 23 Constraint: If the optional characters UNIT= are omitted from the unit specifier, the unit specifier  
 24 must be the first item in the *connect-spec-list*.
- 25 Constraint: Each specifier must not appear more than once in a given *open-stmt*; an *external-file-*  
 26 *unit* must be specified.
- 27 Constraint: The *label* used in the ERR= specifier must be the statement label of a branch target  
 28 statement that appears in the same scoping unit as the OPEN statement.
- 29 R907 *close-stmt* is CLOSE ( *close-spec-list* )
- 30 R908 *close-spec* is [ UNIT= ] *external-file-unit*  
 31 or IOSTAT= *scalar-default-int-variable*  
 32 or ERR= *label*  
 33 or STATUS= *scalar-default-char-expr*
- 34 Constraint: If the optional characters UNIT= are omitted from the unit specifier, the unit specifier  
 35 must be the first item in the *close-spec-list*.
- 36 Constraint: Each specifier must not appear more than once in a given *close-stmt*; an *external-file-*  
 37 *unit* must be specified.
- 38 Constraint: The *label* used in the ERR= specifier must be the statement label of a branch target  
 39 statement that appears in the same scoping unit as the CLOSE statement.
- 40 R909 *read-stmt* is READ ( *io-control-spec-list* ) [ *input-item-list* ]  
 41 or READ *format* [ , *input-item-list* ]
- 42 R910 *write-stmt* is WRITE ( *io-control-spec-list* ) [ *output-item-list* ]
- 43 R911 *print-stmt* is PRINT *format* [ , *output-item-list* ]
- 44 R912 *io-control-spec* is [ UNIT= ] *io-unit*  
 45 or [ FMT= ] *format*  
 46 or [ NML= ] *namelist-group-name*  
 47 or REC= *scalar-int-expr*

- 1 or IOSTAT= *scalar-default-int-variable*  
 2 or ERR= *label*  
 3 or END= *label*  
 4 or ADVANCE= *scalar-default-char-expr*  
 5 or SIZE= *scalar-default-int-variable*  
 6 or EOR= *label*
- 7 Constraint: An *io-control-spec-list* must contain exactly one *io-unit* and may contain at most one of  
 8 each of the other specifiers.
- 9 Constraint: An END=, EOR=, or SIZE= specifier must not appear in a *write-stmt*.
- 10 Constraint: The *label* in the ERR=, EOR=, or END= specifier must be the statement label of a  
 11 branch target statement that appears in the same scoping unit as the data transfer  
 12 statement.
- 13 Constraint: A *namelist-group-name* must not be present if an *input-item-list* or an *output-item-list* is  
 14 present in the data transfer statement.
- 15 Constraint: An *io-control-spec-list* must not contain both a *format* and a *namelist-group-name*.
- 16 Constraint: If the optional characters UNIT= are omitted from the unit specifier, the unit specifier  
 17 must be the first item in the control information list.
- 18 Constraint: If the optional characters FMT= are omitted from the format specifier, the format  
 19 specifier must be the second item in the control information list and the first item  
 20 must be the unit specifier without the optional characters UNIT=.
- 21 Constraint: If the optional characters NML= are omitted from the namelist specifier, the namelist  
 22 specifier must be the second item in the control information list and the first item  
 23 must be the unit specifier without the optional characters UNIT=.
- 24 Constraint: If the unit specifier specifies an internal file, the *io-control-spec-list* must not contain a  
 25 REC= specifier or a *namelist-group-name*.
- 26 Constraint: If the REC= specifier is present, an END= specifier must not appear, a *namelist-*  
 27 *group-name* must not appear, and the *format*, if any, must not be an asterisk specifying  
 28 list-directed input/output.
- 29 Constraint: An ADVANCE= specifier may be present only in a formatted sequential  
 30 input/output statement with explicit format specification (10.1) whose control infor-  
 31 mation list does not contain an internal file unit specifier.
- 32 Constraint: If an EOR= specifier is present, and ADVANCE= specifier also must appear.
- 33 R913 *format* is *default-char-expr*  
 34 or *label*  
 35 or \*  
 36 or *scalar-default-int-variable*
- 37 Constraint: The *label* must be the label of a FORMAT statement that appears in the same scoping  
 38 unit as the statement containing the format specifier.
- 39 R914 *input-item* is *variable*  
 40 or *io-implied-do*
- 41 R915 *output-item* is *expr*  
 42 or *io-implied-do*
- 43 R916 *io-implied-do* is ( *io-implied-do-object-list* , *io-implied-do-control* )
- 44 R917 *io-implied-do-object* is *input-item*  
 45 or *output-item*
- 46 R918 *io-implied-do-control* is *do-variable* = *scalar-numeric-expr* , ■  
 47 ■ *scalar-numeric-expr* [ , *scalar-numeric-expr* ]

## SYNTAX RULES

- 1 Constraint: A *variable* that is an *input-item* must not be an assumed-size array.
- 2 Constraint: The *do-variable* must be a scalar of type integer, default real, or double precision real.
- 3 Constraint: Each *scalar-numeric-expr* in an *io-implied-do-control* must be of type integer, default real,  
4 or double precision real.
- 5 Constraint: In an *input-item-list*, an *io-implied-do-object* must be an *input-item*. In an *output-item-*  
6 *list*, an *io-implied-do-object* must be an *output-item*.
- 7 R919 *backspace-stmt* is BACKSPACE *external-file-unit*  
8 or BACKSPACE ( *position-spec-list* )
- 9 R920 *endfile-stmt* is ENDFILE *external-file-unit*  
10 or ENDFILE ( *position-spec-list* )
- 11 R921 *rewind-stmt* is REWIND *external-file-unit*  
12 or REWIND ( *position-spec-list* )
- 13 R922 *position-spec* is [ UNIT = ] *external-file-unit*  
14 or IOSTAT = *scalar-default-int-variable*  
15 or ERR = *label*
- 16 Constraint: The *label* in the ERR= specifier must be the statement label of a branch target state-  
17 ment that appears in the same scoping unit as the file positioning statement.
- 18 Constraint: If the optional characters UNIT= are omitted from the unit specifier, the unit specifier  
19 must be the first item in the *position-spec-list*.
- 20 Constraint: A *position-spec-list* must contain exactly one *external-file-unit* and may contain at most  
21 one of each of the other specifiers.
- 22 R923 *inquire-stmt* is INQUIRE ( *inquire-spec-list* )  
23 or INQUIRE ( IOLENGTH = *scalar-default-int-variable* ) ■  
24 ■ *output-item-list*
- 25 R924 *inquire-spec* is [ UNIT = ] *external-file-unit*  
26 or FILE = *file-name-expr*  
27 or IOSTAT = *scalar-default-int-variable*  
28 or ERR = *label*  
29 or EXIST = *scalar-default-logical-variable*  
30 or OPENED = *scalar-default-logical-variable*  
31 or NUMBER = *scalar-default-int-variable*  
32 or NAMED = *scalar-default-logical-variable*  
33 or NAME = *scalar-default-char-variable*  
34 or ACCESS = *scalar-default-char-variable*  
35 or SEQUENTIAL = *scalar-default-char-variable*  
36 or DIRECT = *scalar-default-char-variable*  
37 or FORM = *scalar-default-char-variable*  
38 or FORMATTED = *scalar-default-char-variable*  
39 or UNFORMATTED = *scalar-default-char-variable*  
40 or RECL = *scalar-default-int-variable*  
41 or NEXTREC = *scalar-default-int-variable*  
42 or BLANK = *scalar-default-char-variable*  
43 or POSITION = *scalar-default-char-variable*  
44 or ACTION = *scalar-default-char-variable*  
45 or READ = *scalar-default-char-variable*  
46 or WRITE = *scalar-default-char-variable*  
47 or READWRITE = *scalar-default-char-variable*  
48 or DELIM = *scalar-default-char-variable*  
49 or PAD = *scalar-default-char-variable*
- 50 Constraint: An *inquire-spec-list* must contain one FILE= specifier or one UNIT= specifier, but not  
51 both, and at most one of each of the other specifiers.

1 Constraint: In the inquire by unit form of the INQUIRE statement, if the optional characters  
 2 UNIT= are omitted from the unit specifier, the unit specifier must be the first item in  
 3 the *inquire-spec-list*.

#### 4 D.1.10 Input/Output Editing.

5 R1001 *format-stmt* is FORMAT *format-specification*

6 R1002 *format-specification* is ([ *format-item-list* ])

7 Constraint: The *format-stmt* must be labeled.

8 Constraint: The comma used to separate *format-items* in a *format-item-list* may be omitted as fol-  
 9 lows:

10 (1) Between a P edit descriptor and an immediately following F, E, EN, ES, D,  
 11 or G edit descriptor (10.6.5)

12 (2) Before a slash edit descriptor when the optional repeat specification is not  
 13 present (10.6.2)

14 (3) After a slash edit descriptor

15 (4) Before or after a colon edit descriptor (10.6.3)

16 R1003 *format-item* is [ *r* ] *data-edit-desc*  
 17 or *control-edit-desc*  
 18 or *char-string-edit-desc*  
 19 or [ *r* ] ( *format-item-list* )

20 R1004 *r* is *int-literal-constant*

21 Constraint: *r* must be positive.

22 Constraint: *r* must not have a kind parameter specified for it.

23 R1005 *data-edit-desc* is I *w* [ . *m* ]  
 24 or B *w* [ . *m* ]  
 25 or O *w* [ . *m* ]  
 26 or Z *w* [ . *m* ]  
 27 or F *w* . *d*  
 28 or E *w* . *d* [ E *e* ]  
 29 or EN *w* . *d* [ E *e* ]  
 30 or ES *w* . *d* [ E *e* ]  
 31 or G *w* . *d* [ E *e* ]  
 32 or L *w*  
 33 or A [ *w* ]  
 34 or D *w* . *d*

35 R1006 *w* is *int-literal-constant*

36 R1007 *m* is *int-literal-constant*

37 R1008 *d* is *int-literal-constant*

38 R1009 *e* is *int-literal-constant*

39 Constraint: *w* and *e* must be positive.

40 Constraint: *w*, *m*, *d*, and *e* must not have kind parameters specified for them.

41 R1010 *control-edit-desc* is *position-edit-desc*  
 42 or [ *r* ] /  
 43 or :  
 44 or *sign-edit-desc*  
 45 or *kP*  
 46 or *blank-interp-edit-desc*

## SYNTAX RULES

- 1 R1011 *k* is *signed-int-literal-constant*
- 2 Constraint: *k* must not have a kind parameter specified for it.
- 3 R1012 *position-edit-desc* is T *n*
- 4 or TL *n*
- 5 or TR *n*
- 6 or *n* X
- 7 R1013 *n* is *int-literal-constant*
- 8 Constraint: *n* must be positive.
- 9 Constraint: *n* must not have a kind parameter specified for it.
- 10 R1014 *sign-edit-desc* is S
- 11 or SP
- 12 or SS
- 13 R1015 *blank-interp-edit-desc* is BN
- 14 or BZ
- 15 R1016 *char-string-edit-desc* is *char-literal-constant*
- 16 or *c* H *rep-char* [ *rep-char* ] ...
- 17 R1017 *c* is *int-literal-constant*
- 18 Constraint: *c* must be positive.
- 19 Constraint: *c* must not have a kind parameter specified for it.
- 20 Constraint: The *rep-char* in the *cH* form must be of default character type.
- 21 Constraint: The *char-literal-constant* must not have a kind parameter specified for it.
- 22 **D.1.11 Program Units.**
- 23 R1101 *main-program* is [ *program-stmt*
- 24 [ *specification-part*
- 25 [ *execution-part*
- 26 [ *internal-subprogram-part* ]
- 27 *end-program-stmt*
- 28 R1102 *program-stmt* is PROGRAM *program-name*
- 29 R1103 *end-program-stmt* is END [ PROGRAM [ *program-name* ] ]
- 30 Constraint: In a *main-program*, the *execution-part* must not contain a RETURN statement or an
- 31 ENTRY statement.
- 32 Constraint: The *program-name* may be included in the *end-program-stmt* only if the optional
- 33 *program-stmt* is used and, if included, must be identical to the *program-name* specified
- 34 in the *program-stmt*.
- 35 Constraint: An automatic object must not appear in the *specification-part* (R204) of a main pro-
- 36 gram.
- 37 R1104 *module* is *module-stmt*
- 38 [ *specification-part* ]
- 39 [ *module-subprogram-part* ]
- 40 *end-module-stmt*
- 41 R1105 *module-stmt* is MODULE *module-name*
- 42 R1106 *end-module-stmt* is END [ MODULE [ *module-name* ] ]
- 43 Constraint: If the *module-name* is specified in the *end-module-stmt*, it must be identical to the
- 44 *module-name* specified in the *module-stmt*.

- 1 Constraint: A module *specification-part* must not contain a *stmt-function-stmt* or a *format-stmt*.
- 2 Constraint: An automatic object must not appear in the *specification-part* (R204) of a module.
- 3 R1107 *use-stmt* is USE *module-name* [ , *rename-list* ]  
 4 or USE *module-name* , ONLY : [ *only-list* ]
- 5 R1108 *rename* is *local-name* => *use-name*
- 6 R1109 *only* is *access-id*  
 7 or [ *local-name* => ] *use-name*
- 8 Constraint: Each *access-id* must be a public entity in the module.
- 9 Constraint: Each *use-name* must be the name of a public entity in the module.
- 10 R1110 *block-data* is *block-data-stmt*  
 11 [ *specification-part* ]  
 12 *end-block-data-stmt*
- 13 R1111 *block-data-stmt* is BLOCK DATA [ *block-data-name* ]
- 14 R1112 *end-block-data-stmt* is END [ BLOCK DATA [ *block-data-name* ] ]
- 15 Constraint: The *block-data-name* may be included in the *end-block-data-stmt* only if it was provided  
 16 in the *block-data-stmt* and, if included, must be identical to the *block-data-name* in the  
 17 *block-data-stmt*.
- 18 Constraint: A *block-data specification-part* may contain only USE statements, type declaration state-  
 19 ments, IMPLICIT statements, PARAMETER statements, derived-type definitions,  
 20 and the following specification statements: COMMON, DATA, DIMENSION,  
 21 EQUIVALENCE, INTRINSIC, POINTER, SAVE, and TARGET.
- 22 Constraint: A type declaration statement in a *block-data specification-part* must not contain ALLO-  
 23 CATABLE, EXTERNAL, INTENT, OPTIONAL, PRIVATE, or PUBLIC attribute spec-  
 24 ifiers.
- 25 **D.1.12 Procedures.**
- 26 R1201 *interface-block* is *interface-stmt*  
 27 [ *interface-body* ] ...  
 28 [ *module-procedure-stmt* ] ...  
 29 *end-interface-stmt*
- 30 R1202 *interface-stmt* is INTERFACE [ *generic-spec* ]
- 31 R1203 *end-interface-stmt* is END INTERFACE
- 32 R1204 *interface-body* is *function-stmt*  
 33 [ *specification-part* ]  
 34 *end-function-stmt*  
 35 or *subroutine-stmt*  
 36 [ *specification-part* ]  
 37 *end-subroutine-stmt*
- 38 R1205 *module-procedure-stmt* is MODULE PROCEDURE *procedure-name-list*
- 39 R1206 *generic-spec* is *generic-name*  
 40 or OPERATOR ( *defined-operator* )  
 41 or ASSIGNMENT ( = )
- 42 Constraint: An *interface-body* must not contain an *entry-stmt*, *data-stmt*, *format-stmt*, or *stmt-*  
 43 *function-stmt*.
- 44 Constraint: The MODULE PROCEDURE specification is allowed only if the *interface-block* has a  
 45 *generic-spec*.

## SYNTAX RULES

- 1 Constraint: An *interface-block* must not appear in a BLOCK DATA program unit.
- 2 Constraint: An *interface-block* in a subprogram must not contain an *interface-body* for a procedure  
3 defined by that subprogram.
- 4 R1207 *external-stmt* is EXTERNAL *external-name-list*
- 5 R1208 *intrinsic-stmt* is INTRINSIC *intrinsic-procedure-name-list*
- 6 Constraint: Each *intrinsic-procedure-name* must be the name of an intrinsic procedure.
- 7 R1209 *function-reference* is *function-name* ( [ *actual-arg-spec-list* ] )
- 8 Constraint: The *actual-arg-spec-list* for a function reference must not contain an *alt-return-spec*.
- 9 R1210 *call-stmt* is CALL *subroutine-name* [ ( [ *actual-arg-spec-list* ] ) ]
- 10 R1211 *actual-arg-spec* is [ *keyword* = ] *actual-arg*
- 11 R1212 *keyword* is *dummy-arg-name*
- 12 R1213 *actual-arg* is *expr*  
13 or *variable*  
14 or *procedure-name*  
15 or *alt-return-spec*
- 16 R1214 *alt-return-spec* is \* *label*
- 17 Constraint: The *keyword* = must not appear if the interface of the procedure is implicit in the  
18 scoping unit.
- 19 Constraint: The *keyword* = may be omitted from an *actual-arg-spec* only if the *keyword* = has been  
20 omitted from each preceding *actual-arg-spec* in the argument list.
- 21 Constraint: Each *keyword* must be the name of a dummy argument in the explicit interface of the  
22 procedure.
- 23 Constraint: A *procedure-name actual-arg* must not be the name of an internal procedure and must  
24 not be the generic name of a procedure (12.3.2.1, 13.1).
- 25 Constraint: The *label* used in the *alt-return-spec* must be the statement label of a branch target statement that appears in  
26 the same scoping unit as the *call-stmt*.
- 27 R1215 *function-subprogram* is *function-stmt*  
28 [ *specification-part* ]  
29 [ *execution-part* ]  
30 [ *internal-subprogram-part* ]  
31 *end-function-stmt*
- 32 R1216 *function-stmt* is [ *prefix* ] FUNCTION *function-name* ■  
33 ■ ( [ *dummy-arg-name-list* ] ) [ RESULT ( *result-name* ) ]
- 34 Constraint: The *function-name*, if different from the *result-name*, must not appear in any specifica-  
35 tion statement in the scoping unit of the function subprogram.
- 36 R1217 *prefix* is *type-spec* [ RECURSIVE ]  
37 or RECURSIVE [ *type-spec* ]
- 38 R1218 *end-function-stmt* is END [ FUNCTION [ *function-name* ] ]
- 39 Constraint: If RESULT is specified, *result-name* must not be the same as *function-name*.
- 40 Constraint: FUNCTION must be present on the *end-function-stmt* of an internal or module func-  
41 tion.
- 42 Constraint: An internal function must not contain an ENTRY statement.
- 43 Constraint: An internal function must not contain an *internal-subprogram-part*.



- 1 Constraint: If a *function-name* is present on the *end-function-stmt*, it must be identical to the  
2 *function-name* specified in the *function-stmt*.
- 3 R1219 *subroutine-subprogram* is *subroutine-stmt*  
4 [ *specification-part* ]  
5 [ *execution-part* ]  
6 [ *internal-subprogram-part* ]  
7 *end-subroutine-stmt*
- 8 R1220 *subroutine-stmt* is [ RECURSIVE ] SUBROUTINE *subroutine-name* ■  
9 ■ [ ( [ *dummy-arg-list* ] ) ]
- 10 R1221 *dummy-arg* is *dummy-arg-name*  
11 or \*
- 12 R1222 *end-subroutine-stmt* is END [ SUBROUTINE [ *subroutine-name* ] ]
- 13 Constraint: SUBROUTINE must be present on the *end-subroutine-stmt* of an internal or module  
14 subroutine.
- 15 Constraint: An internal subroutine must not contain an ENTRY statement.
- 16 Constraint: An internal subroutine must not contain an *internal-subprogram-part*.
- 17 Constraint: If a *subroutine-name* is present on the *end-subroutine-stmt*, it must be identical to the  
18 *subroutine-name* specified in the *subroutine-stmt*.
- 19 R1223 *entry-stmt* is ENTRY *entry-name* [ ( [ *dummy-arg-list* ] ) ] ■  
20 ■ [ RESULT ( *result-name* ) ]
- 21 Constraint: An *entry-stmt* may appear only in an *external-subprogram* or *module-subprogram*. An  
22 *entry-stmt* must not appear within an *executable-construct*.
- 23 Constraint: RESULT may be present only if the *entry-stmt* is contained in a function subprogram.
- 24 Constraint: Within the subprogram containing the *entry-stmt*, the *entry-name* must not appear as  
25 a dummy argument in the FUNCTION or SUBROUTINE statement or in another  
26 ENTRY statement and it must not appear in an EXTERNAL statement.
- 27 Constraint: A *dummy-arg* may be an alternate return indicator only if the ENTRY statement is contained in a subrou-  
28 tine subprogram.
- 29 Constraint: If RESULT is specified, *result-name* must not be the same as *entry-name*.
- 30 R1224 *return-stmt* is RETURN [ *scalar-int-expr* ]
- 31 Constraint: The *return-stmt* must be contained in the scoping unit of a function or subroutine sub-  
32 program.
- 33 Constraint: The *scalar-int-expr* is allowed only in the scoping unit of a subroutine subprogram.
- 34 R1225 *contains-stmt* is CONTAINS
- 35 R1226 *stmt-function-stmt* is *function-name* ( [ *dummy-arg-name-list* ] ) = *scalar-expr*
- 36 Constraint: The *scalar-expr* may be composed only of constants (literal and named), references to  
37 scalar variables and array elements, references to functions and function dummy  
38 procedures, and intrinsic operators. If a reference to another statement function  
39 appears in *scalar-expr*, its definition must have been provided earlier in the scoping  
40 unit.
- 41 Constraint: Named constants in *scalar-expr* must have been declared earlier in the scoping unit. If  
42 array elements appear in *scalar-expr*, the parent array must have been declared as an  
43 array earlier in the scoping unit. If a scalar variable, array element, function refer-  
44 ence, or dummy function reference is typed by the implicit typing rules, its appear-  
45 ance in any subsequent type declaration statement must confirm this implied type  
46 and the values of any implied type parameters.

## SYNTAX RULES

- 1 Constraint: The *function-name* and each *dummy-arg-name* must be specified, explicitly or implicitly, to be scalar data objects.
- 2
- 3 Constraint: A given *dummy-arg-name* may appear only once in any *dummy-arg-name-list*.
- 4 Constraint: Each scalar variable reference in *scalar-expr* 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.
- 5
- 6

### 7 D.1.13 Intrinsic Procedures.

### 8 D.1.14 Scope, Association, and Definition.

- 9 **D.2 Cross References.** In the following cross references, all occurrences of *scalar-xyz* and *xyz-list* are treated as occurrences of *xyz*.
- 10

### 11 D.2.1 Nonterminal Symbols That Are Defined.

| 12 | Symbol                          | Defined in | Referenced in |       |      |      |
|----|---------------------------------|------------|---------------|-------|------|------|
| 13 | <i>ac-do-variable</i>           | R435       | R434          |       |      |      |
| 14 | <i>ac-implied-do</i>            | R433       | R432          |       |      |      |
| 15 | <i>ac-implied-do-control</i>    | R434       | R433          |       |      |      |
| 16 | <i>ac-value</i>                 | R432       | R431          | R433  |      |      |
| 17 | <i>access-id</i>                | R522       | R521          | R1109 |      |      |
| 18 | <i>access-spec</i>              | R510       | R424          | R503  | R521 |      |
| 19 | <i>access-stmt</i>              | R521       | R214          |       |      |      |
| 20 | <i>action-stmt</i>              | R216       | R215          | R807  | R829 | R833 |
| 21 | <i>action-term-do-construct</i> | R827       | R826          |       |      |      |
| 22 | <i>actual-arg</i>               | R1213      | R1211         |       |      |      |
| 23 | <i>actual-arg-spec</i>          | R1211      | R1209         | R1210 |      |      |
| 24 | <i>add-op</i>                   | R710       | R310          | R707  |      |      |
| 25 | <i>add-operand</i>              | R706       | R706          | R707  |      |      |
| 26 | <i>allocatable-stmt</i>         | R526       | R214          |       |      |      |
| 27 | <i>allocate-object</i>          | R625       | R624          | R628  |      |      |
| 28 | <i>allocate-stmt</i>            | R622       | R216          |       |      |      |
| 29 | <i>allocation</i>               | R624       | R622          |       |      |      |
| 30 | <i>alphanumeric-character</i>   | R302       | R301          | R304  |      |      |
| 31 | <i>alt-return-spec</i>          | R1214      | R1213         |       |      |      |
| 32 | <i>and-op</i>                   | R720       | R310          | R716  |      |      |
| 33 | <i>and-operand</i>              | R715       | R716          |       |      |      |
| 34 | <i>arithmetic-if-stmt</i>       | R840       | R216          |       |      |      |
| 35 | <i>array-constructor</i>        | R431       | R701          |       |      |      |
| 36 | <i>array-element</i>            | R615       | R536          | R547  | R602 | R610 |
| 37 | <i>array-section</i>            | R616       | R602          |       |      |      |
| 38 | <i>array-spec</i>               | R512       | R503          | R504  | R525 | R528 |
| 39 | <i>assign-stmt</i>              | R838       | R216          |       |      |      |
| 40 | <i>assigned-goto-stmt</i>       | R839       | R216          |       |      |      |
| 41 | <i>assignment-stmt</i>          | R735       | R216          | R738  | R739 |      |
| 42 | <i>assumed-shape-spec</i>       | R516       | R512          |       |      |      |
| 43 | <i>assumed-size-spec</i>        | R518       | R512          |       |      |      |
| 44 | <i>attr-spec</i>                | R503       | R501          |       |      |      |
| 45 | <i>backspace-stmt</i>           | R919       | R216          |       |      |      |
| 46 | <i>binary-constant</i>          | R408       | R407          |       |      |      |
| 47 | <i>blank-interp-edit-desc</i>   | R1015      | R1010         |       |      |      |
| 48 | <i>block</i>                    | R801       | R802          | R808  | R823 |      |
| 49 | <i>block-data</i>               | R1110      | R202          |       |      |      |
| 50 | <i>block-data-stmt</i>          | R1111      | R1110         |       |      |      |

|    | Symbol                          | Defined in | Referenced in |       |      |           |
|----|---------------------------------|------------|---------------|-------|------|-----------|
| 1  | <i>block-do-construct</i>       | R817       | R816          |       |      |           |
| 2  | <i>boz-literal-constant</i>     | R407       | R306          | R533  |      |           |
| 3  | <i>c</i>                        | R1017      | R1016         |       |      |           |
| 4  | <i>call-stmt</i>                | R1210      | R216          |       |      |           |
| 5  | <i>case-construct</i>           | R808       | R215          |       |      |           |
| 6  | <i>case-expr</i>                | R812       | R809          |       |      |           |
| 7  | <i>case-selector</i>            | R813       | R810          |       |      |           |
| 8  | <i>case-stmt</i>                | R810       | R808          |       |      |           |
| 9  | <i>case-value</i>               | R815       | R814          |       |      |           |
| 10 | <i>case-value-range</i>         | R814       | R813          |       |      |           |
| 11 | <i>char-constant</i>            | R309       | R843          |       |      |           |
| 12 | <i>char-expr</i>                | R726       | R731          | R812  |      |           |
| 13 | <i>char-initialization-expr</i> | R731       | R815          |       |      |           |
| 14 | <i>char-length</i>              | R508       | R429          | R504  | R507 |           |
| 15 | <i>char-literal-constant</i>    | R420       | R306          | R1016 |      |           |
| 16 | <i>char-selector</i>            | R506       | R502          |       |      |           |
| 17 | <i>char-string-edit-desc</i>    | R1016      | R1003         |       |      |           |
| 18 | <i>char-variable</i>            | R605       |               |       |      |           |
| 19 | <i>character</i>                | R301       |               |       |      |           |
| 20 | <i>close-spec</i>               | R908       | R907          |       |      |           |
| 21 | <i>close-stmt</i>               | R907       | R216          |       |      |           |
| 22 | <i>common-block-object</i>      | R549       | R548          |       |      |           |
| 23 | <i>common-stmt</i>              | R548       | R214          |       |      |           |
| 24 | <i>complex-literal-constant</i> | R417       | R306          |       |      |           |
| 25 | <i>component-array-spec</i>     | R428       | R427          | R429  |      |           |
| 26 | <i>component-attr-spec</i>      | R427       | R426          |       |      |           |
| 27 | <i>component-decl</i>           | R429       | R426          |       |      |           |
| 28 | <i>component-def-stmt</i>       | R426       | R422          |       |      |           |
| 29 | <i>computed-goto-stmt</i>       | R837       | R216          |       |      |           |
| 30 | <i>concat-op</i>                | R712       | R310          | R711  |      |           |
| 31 | <i>connect-spec</i>             | R905       | R904          |       |      |           |
| 32 | <i>constant</i>                 | R305       | R308          | R309  | R533 | R610 R701 |
| 33 | <i>constant-subobject</i>       | R702       | R701          |       |      |           |
| 34 | <i>contains-stmt</i>            | R1225      | R210          | R212  |      |           |
| 35 | <i>continue-stmt</i>            | R841       | R216          | R824  |      |           |
| 36 | <i>control-edit-desc</i>        | R1010      | R1003         |       |      |           |
| 37 | <i>cycle-stmt</i>               | R834       | R216          |       |      |           |
| 38 | <i>d</i>                        | R1008      | R1005         |       |      |           |
| 39 | <i>data-edit-desc</i>           | R1005      | R1003         |       |      |           |
| 40 | <i>data-i-do-object</i>         | R536       | R535          |       |      |           |
| 41 | <i>data-i-do-variable</i>       | R537       | R535          |       |      |           |
| 42 | <i>data-implied-do</i>          | R535       | R531          | R536  |      |           |
| 43 | <i>data-ref</i>                 | R612       | R614          | R615  | R616 |           |
| 44 | <i>data-stmt</i>                | R529       | R209          | R214  |      |           |
| 45 | <i>data-stmt-constant</i>       | R533       | R532          |       |      |           |
| 46 | <i>data-stmt-object</i>         | R531       | R530          |       |      |           |
| 47 | <i>data-stmt-repeat</i>         | R534       | R532          |       |      |           |
| 48 | <i>data-stmt-set</i>            | R530       | R529          |       |      |           |
| 49 | <i>data-stmt-value</i>          | R532       | R530          |       |      |           |
| 50 | <i>deallocate-stmt</i>          | R628       | R216          |       |      |           |
| 51 | <i>declaration-construct</i>    | R207       | R204          |       |      |           |
| 52 | <i>default-char-expr</i>        | R727       | R905          | R906  | R908 | R912 R913 |
| 53 | <i>default-char-variable</i>    | R606       | R903          | R924  |      |           |
| 54 | <i>default-int-variable</i>     | R608       | R905          | R908  | R912 | R913 R922 |
| 55 |                                 |            | R923          | R924  |      |           |
| 56 | <i>default-logical-variable</i> | R604       | R924          |       |      |           |

## SYNTAX RULES

|    | Symbol                          | Defined in | Referenced in |       |       |       |       |
|----|---------------------------------|------------|---------------|-------|-------|-------|-------|
| 1  | <i>deferred-shape-spec</i>      | R517       | R428          | R512  | R526  | R527  |       |
| 2  | <i>defined-binary-op</i>        | R724       | R311          | R723  |       |       |       |
| 3  | <i>defined-operator</i>         | R311       | R1206         |       |       |       |       |
| 4  | <i>defined-unary-op</i>         | R704       | R311          | R703  |       |       |       |
| 5  | <i>derived-type-def</i>         | R422       | R207          |       |       |       |       |
| 6  | <i>derived-type-stmt</i>        | R424       | R422          |       |       |       |       |
| 7  | <i>digit-string</i>             | R402       | R401          | R404  | R405  | R413  | R414  |
| 8  | <i>dimension-stmt</i>           | R525       | R214          |       |       |       |       |
| 9  | <i>do-block</i>                 | R823       | R817          |       |       |       |       |
| 10 | <i>do-body</i>                  | R828       | R827          | R830  | R832  |       |       |
| 11 | <i>do-construct</i>             | R816       | R215          |       |       |       |       |
| 12 | <i>do-stmt</i>                  | R818       | R817          |       |       |       |       |
| 13 | <i>do-term-action-stmt</i>      | R829       | R827          |       |       |       |       |
| 14 | <i>do-term-shared-stmt</i>      | R833       | R832          |       |       |       |       |
| 15 | <i>do-variable</i>              | R822       | R821          | R918  |       |       |       |
| 16 | <i>dummy-arg</i>                | R1221      | R1220         | R1223 |       |       |       |
| 17 | <i>e</i>                        | R1009      | R1005         |       |       |       |       |
| 18 | <i>else-if-stmt</i>             | R804       | R802          |       |       |       |       |
| 19 | <i>else-stmt</i>                | R805       | R802          |       |       |       |       |
| 20 | <i>elsewhere-stmt</i>           | R742       | R739          |       |       |       |       |
| 21 | <i>end-block-data-stmt</i>      | R1112      | R1110         |       |       |       |       |
| 22 | <i>end-do</i>                   | R824       | R817          |       |       |       |       |
| 23 | <i>end-do-stmt</i>              | R825       | R824          |       |       |       |       |
| 24 | <i>end-function-stmt</i>        | R1218      | R216          | R1204 | R1215 |       |       |
| 25 | <i>end-if-stmt</i>              | R806       | R802          |       |       |       |       |
| 26 | <i>end-interface-stmt</i>       | R1203      | R1201         |       |       |       |       |
| 27 | <i>end-module-stmt</i>          | R1106      | R1104         |       |       |       |       |
| 28 | <i>end-program-stmt</i>         | R1103      | R216          | R1101 |       |       |       |
| 29 | <i>end-select-stmt</i>          | R811       | R808          |       |       |       |       |
| 30 | <i>end-subroutine-stmt</i>      | R1222      | R216          | R1204 | R1219 |       |       |
| 31 | <i>end-type-stmt</i>            | R425       | R422          |       |       |       |       |
| 32 | <i>end-where-stmt</i>           | R743       | R739          |       |       |       |       |
| 33 | <i>endfile-stmt</i>             | R920       | R216          |       |       |       |       |
| 34 | <i>entity-decl</i>              | R504       | R501          |       |       |       |       |
| 35 | <i>entry-stmt</i>               | R1223      | R206          | R207  | R209  |       |       |
| 36 | <i>equiv-op</i>                 | R722       | R310          | R718  |       |       |       |
| 37 | <i>equiv-operand</i>            | R717       | R717          | R718  |       |       |       |
| 38 | <i>equivalence-object</i>       | R547       | R546          |       |       |       |       |
| 39 | <i>equivalence-set</i>          | R546       | R545          |       |       |       |       |
| 40 | <i>equivalence-stmt</i>         | R545       | R214          |       |       |       |       |
| 41 | <i>executable-construct</i>     | R215       | R208          | R209  |       |       |       |
| 42 | <i>executable-program</i>       | R201       |               |       |       |       |       |
| 43 | <i>execution-part</i>           | R208       | R1101         | R1215 | R1219 |       |       |
| 44 | <i>execution-part-construct</i> | R209       | R208          | R801  | R828  |       |       |
| 45 | <i>exit-stmt</i>                | R835       | R216          |       |       |       |       |
| 46 | <i>explicit-shape-spec</i>      | R513       | R428          | R512  | R518  | R549  | R624  |
| 47 | <i>exponent</i>                 | R416       | R413          |       |       |       |       |
| 48 | <i>exponent-letter</i>          | R415       | R413          |       |       |       |       |
| 49 | <i>expr</i>                     | R723       | R430          | R432  | R701  | R723  | R725  |
| 50 |                                 |            | R726          | R727  | R728  | R729  | R730  |
| 51 |                                 |            | R735          | R737  | R915  | R1213 | R1226 |
| 52 | <i>extended-intrinsic-op</i>    | R312       | R311          |       |       |       |       |
| 53 | <i>external-file-unit</i>       | R902       | R901          | R905  | R908  | R919  | R920  |
| 54 |                                 |            | R921          | R922  | R924  |       |       |
| 55 | <i>external-stmt</i>            | R1207      | R214          |       |       |       |       |
| 56 | <i>external-subprogram</i>      | R203       | R202          |       |       |       |       |

|    | Symbol                           | Defined in | Referenced in |       |       |       |       |
|----|----------------------------------|------------|---------------|-------|-------|-------|-------|
| 1  | <i>file-name-expr</i>            | R906       | R905          | R924  |       |       |       |
| 2  | <i>format</i>                    | R913       | R909          | R911  | R912  |       |       |
| 3  | <i>format-item</i>               | R1003      | R1002         | R1003 |       |       |       |
| 4  | <i>format-specification</i>      | R1002      | R1001         |       |       |       |       |
| 5  | <i>format-stmt</i>               | R1001      | R206          | R207  | R209  |       |       |
| 6  | <i>function-reference</i>        | R1209      | R701          |       |       |       |       |
| 7  | <i>function-stmt</i>             | R1216      | R1204         | R1215 |       |       |       |
| 8  | <i>function-subprogram</i>       | R1215      | R203          | R211  | R213  |       |       |
| 9  | <i>generic-spec</i>              | R1206      | R522          | R1202 |       |       |       |
| 10 | <i>goto-stmt</i>                 | R836       | R216          |       |       |       |       |
| 11 | <i>hex-constant</i>              | R410       | R407          |       |       |       |       |
| 12 | <i>hex-digit</i>                 | R411       | R410          |       |       |       |       |
| 13 | <i>if-construct</i>              | R802       | R215          |       |       |       |       |
| 14 | <i>if-stmt</i>                   | R807       | R216          |       |       |       |       |
| 15 | <i>if-then-stmt</i>              | R803       | R802          |       |       |       |       |
| 16 | <i>imag-part</i>                 | R419       | R417          |       |       |       |       |
| 17 | <i>implicit-part</i>             | R205       | R204          |       |       |       |       |
| 18 | <i>implicit-part-stmt</i>        | R206       | R205          |       |       |       |       |
| 19 | <i>implicit-spec</i>             | R541       | R540          |       |       |       |       |
| 20 | <i>implicit-stmt</i>             | R540       | R205          | R206  |       |       |       |
| 21 | <i>initialization-expr</i>       | R730       | R504          | R539  |       |       |       |
| 22 | <i>inner-shared-do-construct</i> | R832       | R831          |       |       |       |       |
| 23 | <i>input-item</i>                | R914       | R909          | R917  |       |       |       |
| 24 | <i>inquire-spec</i>              | R924       | R923          |       |       |       |       |
| 25 | <i>inquire-stmt</i>              | R923       | R216          |       |       |       |       |
| 26 | <i>int-constant</i>              | R308       | R534          |       |       |       |       |
| 27 | <i>int-expr</i>                  | R728       | R434          | R535  | R611  | R617  | R620  |
| 28 |                                  |            | R621          | R732  | R734  | R812  | R837  |
| 29 |                                  |            | R902          | R905  | R912  | R1224 |       |
| 30 | <i>int-initialization-expr</i>   | R732       | R505          | R506  | R815  |       |       |
| 31 | <i>int-literal-constant</i>      | R404       | R306          | R403  | R508  | R1004 | R1006 |
| 32 |                                  |            | R1007         | R1008 | R1009 | R1013 | R1017 |
| 33 | <i>int-variable</i>              | R607       | R435          | R537  | R623  | R838  | R839  |
| 34 | <i>intent-spec</i>               | R511       | R503          | R519  |       |       |       |
| 35 | <i>intent-stmt</i>               | R519       | R214          |       |       |       |       |
| 36 | <i>interface-block</i>           | R1201      | R207          |       |       |       |       |
| 37 | <i>interface-body</i>            | R1204      | R1201         |       |       |       |       |
| 38 | <i>interface-stmt</i>            | R1202      | R1201         |       |       |       |       |
| 39 | <i>internal-file-unit</i>        | R903       | R901          |       |       |       |       |
| 40 | <i>internal-subprogram</i>       | R211       | R210          |       |       |       |       |
| 41 | <i>internal-subprogram-part</i>  | R210       | R1101         | R1215 | R1219 |       |       |
| 42 | <i>intrinsic-operator</i>        | R310       | R312          |       |       |       |       |
| 43 | <i>intrinsic-stmt</i>            | R1208      | R214          |       |       |       |       |
| 44 | <i>io-control-spec</i>           | R912       | R909          | R910  |       |       |       |
| 45 | <i>io-implied-do</i>             | R916       | R914          | R915  |       |       |       |
| 46 | <i>io-implied-do-control</i>     | R918       | R916          |       |       |       |       |
| 47 | <i>io-implied-do-object</i>      | R917       | R916          |       |       |       |       |
| 48 | <i>io-unit</i>                   | R901       | R912          |       |       |       |       |
| 49 | <i>k</i>                         | R1011      | R1010         |       |       |       |       |
| 50 | <i>keyword</i>                   | R1212      | R1211         |       |       |       |       |
| 51 | <i>kind-param</i>                | R405       | R404          | R413  | R420  | R421  |       |
| 52 | <i>kind-selector</i>             | R505       | R502          |       |       |       |       |
| 53 | <i>label</i>                     | R313       | R819          | R836  | R837  | R838  | R839  |
| 54 |                                  |            | R840          | R905  | R908  | R912  | R913  |
| 55 |                                  |            | R922          | R924  | R1214 |       |       |
| 56 | <i>label-do-stmt</i>             | R819       | R818          | R827  | R830  | R832  |       |

**SYNTAX RULES**

|    | Symbol                             | Defined in | Referenced in |      |      |           |
|----|------------------------------------|------------|---------------|------|------|-----------|
| 1  | <i>length-selector</i>             | R507       | R506          |      |      |           |
| 2  | <i>letter-spec</i>                 | R542       | R541          |      |      |           |
| 3  | <i>level-1-expr</i>                | R703       | R705          |      |      |           |
| 4  | <i>level-2-expr</i>                | R707       | R707          | R711 |      |           |
| 5  | <i>level-3-expr</i>                | R711       | R711          | R713 |      |           |
| 6  | <i>level-4-expr</i>                | R713       | R715          |      |      |           |
| 7  | <i>level-5-expr</i>                | R718       | R718          | R723 |      |           |
| 8  | <i>literal-constant</i>            | R306       | R305          |      |      |           |
| 9  | <i>logical-expr</i>                | R725       | R733          | R741 | R803 | R804 R807 |
| 10 |                                    |            | R812          | R821 |      |           |
| 11 | <i>logical-initialization-expr</i> | R733       | R815          |      |      |           |
| 12 | <i>logical-literal-constant</i>    | R421       | R306          |      |      |           |
| 13 | <i>logical-variable</i>            | R603       |               |      |      |           |
| 14 | <i>loop-control</i>                | R821       | R819          | R820 |      |           |
| 15 | <i>lower-bound</i>                 | R514       | R513          | R516 | R518 |           |
| 16 | <i>m</i>                           | R1007      | R1005         |      |      |           |
| 17 | <i>main-program</i>                | R1101      | R202          |      |      |           |
| 18 | <i>mask-expr</i>                   | R741       | R738          | R740 |      |           |
| 19 | <i>module</i>                      | R1104      | R202          |      |      |           |
| 20 | <i>module-procedure-stmt</i>       | R1205      | R1201         |      |      |           |
| 21 | <i>module-stmt</i>                 | R1105      | R1104         |      |      |           |
| 22 | <i>module-subprogram</i>           | R213       | R212          |      |      |           |
| 23 | <i>module-subprogram-part</i>      | R212       | R1104         |      |      |           |
| 24 | <i>mult-op</i>                     | R709       | R310          | R706 |      |           |
| 25 | <i>mult-operand</i>                | R705       | R705          | R706 |      |           |
| 26 | <i>n</i>                           | R1013      | R1012         |      |      |           |
| 27 | <i>name</i>                        | R304       | R307          |      |      |           |
| 28 | <i>named-constant</i>              | R307       | R305          | R539 |      |           |
| 29 | <i>named-constant-def</i>          | R539       | R538          |      |      |           |
| 30 | <i>namelist-group-object</i>       | R544       | R543          |      |      |           |
| 31 | <i>namelist-stmt</i>               | R543       | R214          |      |      |           |
| 32 | <i>nonblock-do-construct</i>       | R826       | R816          |      |      |           |
| 33 | <i>nonlabel-do-stmt</i>            | R820       | R818          |      |      |           |
| 34 | <i>not-op</i>                      | R719       | R310          | R715 |      |           |
| 35 | <i>nullify-stmt</i>                | R626       | R216          |      |      |           |
| 36 | <i>numeric-expr</i>                | R729       | R821          | R840 | R918 |           |
| 37 | <i>octal-constant</i>              | R409       | R407          |      |      |           |
| 38 | <i>only</i>                        | R1109      | R1107         |      |      |           |
| 39 | <i>open-stmt</i>                   | R904       | R216          |      |      |           |
| 40 | <i>optional-stmt</i>               | R520       | R214          |      |      |           |
| 41 | <i>or-op</i>                       | R721       | R310          | R717 |      |           |
| 42 | <i>or-operand</i>                  | R716       | R716          | R717 |      |           |
| 43 | <i>outer-shared-do-construct</i>   | R830       | R826          | R831 |      |           |
| 44 | <i>output-item</i>                 | R915       | R910          | R911 | R917 | R923      |
| 45 | <i>parameter-stmt</i>              | R538       | R206          | R207 |      |           |
| 46 | <i>parent-string</i>               | R610       | R609          |      |      |           |
| 47 | <i>part-ref</i>                    | R613       | R612          |      |      |           |
| 48 | <i>pause-stmt</i>                  | R844       | R216          |      |      |           |
| 49 | <i>pointer-assignment-stmt</i>     | R736       | R216          |      |      |           |
| 50 | <i>pointer-object</i>              | R627       | R626          | R736 |      |           |
| 51 | <i>pointer-stmt</i>                | R527       | R214          |      |      |           |
| 52 | <i>position-edit-desc</i>          | R1012      | R1010         |      |      |           |
| 53 | <i>position-spec</i>               | R922       | R919          | R920 | R921 |           |
| 54 | <i>power-op</i>                    | R708       | R310          | R705 |      |           |
| 55 | <i>prefix</i>                      | R1217      | R1216         |      |      |           |
| 56 | <i>primary</i>                     | R701       | R703          |      |      |           |

|    | Symbol                              | Defined in | Referenced in |       |       |             |
|----|-------------------------------------|------------|---------------|-------|-------|-------------|
| 1  | <i>print-stmt</i>                   | R911       | R216          |       |       |             |
| 2  | <i>private-sequence-stmt</i>        | R423       | R422          |       |       |             |
| 3  | <i>program-stmt</i>                 | R1102      | R1101         |       |       |             |
| 4  | <i>program-unit</i>                 | R202       | R201          |       |       |             |
| 5  | <i>r</i>                            | R1004      | R1003         | R1010 |       |             |
| 6  | <i>read-stmt</i>                    | R909       | R216          |       |       |             |
| 7  | <i>real-literal-constant</i>        | R413       | R306          | R412  |       |             |
| 8  | <i>real-part</i>                    | R418       | R417          |       |       |             |
| 9  | <i>rel-op</i>                       | R714       | R310          | R713  |       |             |
| 10 | <i>rename</i>                       | R1108      | R1107         |       |       |             |
| 11 | <i>return-stmt</i>                  | R1224      | R216          |       |       |             |
| 12 | <i>rewind-stmt</i>                  | R921       | R216          |       |       |             |
| 13 | <i>save-stmt</i>                    | R523       | R214          |       |       |             |
| 14 | <i>saved-entity</i>                 | R524       | R523          |       |       |             |
| 15 | <i>section-subscript</i>            | R618       | R613          |       |       |             |
| 16 | <i>select-case-stmt</i>             | R809       | R808          |       |       |             |
| 17 | <i>shared-term-do-construct</i>     | R831       | R830          |       |       |             |
| 18 | <i>sign</i>                         | R406       | R401          | R403  | R412  |             |
| 19 | <i>sign-edit-desc</i>               | R1014      | R1010         |       |       |             |
| 20 | <i>signed-digit-string</i>          | R401       | R416          |       |       |             |
| 21 | <i>signed-int-literal-constant</i>  | R403       | R418          | R419  | R533  | R1011       |
| 22 | <i>signed-real-literal-constant</i> | R412       | R418          | R419  | R533  |             |
| 23 | <i>significand</i>                  | R414       | R413          |       |       |             |
| 24 | <i>specification-expr</i>           | R734       | R509          | R514  | R515  |             |
| 25 | <i>specification-part</i>           | R204       | R1101         | R1104 | R1110 | R1204 R1215 |
| 26 |                                     |            | R1219         |       |       |             |
| 27 | <i>specification-stmt</i>           | R214       | R207          |       |       |             |
| 28 | <i>stat-variable</i>                | R623       | R622          | R628  |       |             |
| 29 | <i>stmt-function-stmt</i>           | R1226      | R207          |       |       |             |
| 30 | <i>stop-code</i>                    | R843       | R842          | R844  |       |             |
| 31 | <i>stop-stmt</i>                    | R842       | R216          |       |       |             |
| 32 | <i>stride</i>                       | R620       | R619          |       |       |             |
| 33 | <i>structure-component</i>          | R614       | R602          | R610  | R625  | R627        |
| 34 | <i>structure-constructor</i>        | R430       | R533          | R701  |       |             |
| 35 | <i>subobject</i>                    | R602       | R601          | R702  |       |             |
| 36 | <i>subroutine-stmt</i>              | R1220      | R1204         | R1219 |       |             |
| 37 | <i>subroutine-subprogram</i>        | R1219      | R203          | R211  | R213  |             |
| 38 | <i>subscript</i>                    | R617       | R618          | R619  |       |             |
| 39 | <i>subscript-triplet</i>            | R619       | R618          |       |       |             |
| 40 | <i>substring</i>                    | R609       | R547          | R602  |       |             |
| 41 | <i>substring-range</i>              | R611       | R609          | R616  |       |             |
| 42 | <i>target</i>                       | R737       | R736          |       |       |             |
| 43 | <i>target-stmt</i>                  | R528       | R214          |       |       |             |
| 44 | <i>type-declaration-stmt</i>        | R501       | R207          |       |       |             |
| 45 | <i>type-param-value</i>             | R509       | R506          | R507  | R508  |             |
| 46 | <i>type-spec</i>                    | R502       | R426          | R501  | R541  | R1217       |
| 47 | <i>underscore</i>                   | R303       | R302          |       |       |             |
| 48 | <i>upper-bound</i>                  | R515       | R513          |       |       |             |
| 49 | <i>use-stmt</i>                     | R1107      | R204          |       |       |             |
| 50 | <i>variable</i>                     | R601       | R531          | R603  | R604  | R605 R606   |
| 51 |                                     |            | R607          | R608  | R701  | R735 R737   |
| 52 |                                     |            | R822          | R914  | R1213 |             |
| 53 | <i>vector-subscript</i>             | R621       | R618          |       |       |             |
| 54 | <i>w</i>                            | R1006      | R1005         |       |       |             |
| 55 | <i>where-construct</i>              | R739       | R215          |       |       |             |
| 56 | <i>where-construct-stmt</i>         | R740       | R739          |       |       |             |

# SYNTAX RULES

|    | Symbol                                                 | Defined in | Referenced in |       |       |       |       |
|----|--------------------------------------------------------|------------|---------------|-------|-------|-------|-------|
| 1  | <i>where-stmt</i>                                      | R738       | R216          |       |       |       |       |
| 2  | <i>write-stmt</i>                                      | R910       | R216          |       |       |       |       |
| 3  | <b>D.2.2 Nonterminal Symbols That Are Not Defined.</b> |            |               |       |       |       |       |
| 4  | <i>array-name</i>                                      |            | R525          | R526  |       |       |       |
| 5  | <i>array-variable-name</i>                             |            | R601          |       |       |       |       |
| 6  | <i>block-data-name</i>                                 |            | R1111         | R1112 |       |       |       |
| 7  | <i>case-construct-name</i>                             |            | R809          | R810  | R811  |       |       |
| 8  | <i>common-block-name</i>                               |            | R524          | R548  |       |       |       |
| 9  | <i>component-name</i>                                  |            | R429          |       |       |       |       |
| 10 | <i>digit</i>                                           |            | R302          | R313  | R402  | R408  | R409  |
| 11 |                                                        |            | R411          | R843  |       |       |       |
| 12 | <i>do-construct-name</i>                               |            | R819          | R820  | R825  | R834  | R835  |
| 13 | <i>dummy-arg-name</i>                                  |            | R519          | R520  | R1212 | R1216 | R1221 |
| 14 |                                                        |            | R1226         |       |       |       |       |
| 15 | <i>entry-name</i>                                      |            | R1223         |       |       |       |       |
| 16 | <i>external-name</i>                                   |            | R1207         |       |       |       |       |
| 17 | <i>function-name</i>                                   |            | R504          | R1209 | R1216 | R1218 | R1226 |
| 18 | <i>generic-name</i>                                    |            | R1206         |       |       |       |       |
| 19 | <i>if-construct-name</i>                               |            | R803          | R804  | R805  | R806  |       |
| 20 | <i>int-constant-name</i>                               |            | R405          |       |       |       |       |
| 21 | <i>intrinsic-procedure-name</i>                        |            | R1208         |       |       |       |       |
| 22 | <i>letter</i>                                          |            | R302          | R304  | R542  | R704  | R724  |
| 23 | <i>local-name</i>                                      |            | R1108         | R1109 |       |       |       |
| 24 | <i>module-name</i>                                     |            | R1105         | R1106 | R1107 |       |       |
| 25 | <i>namelist-group-name</i>                             |            | R543          | R912  |       |       |       |
| 26 | <i>object-name</i>                                     |            | R504          | R524  | R527  | R528  |       |
| 27 | <i>part-name</i>                                       |            | R613          |       |       |       |       |
| 28 | <i>procedure-name</i>                                  |            | R1205         | R1213 |       |       |       |
| 29 | <i>program-name</i>                                    |            | R1102         | R1103 |       |       |       |
| 30 | <i>rep-char</i>                                        |            | R420          | R1016 |       |       |       |
| 31 | <i>result-name</i>                                     |            | R1216         | R1223 |       |       |       |
| 32 | <i>special-character</i>                               |            | R301          |       |       |       |       |
| 33 | <i>subroutine-name</i>                                 |            | R1210         | R1220 | R1222 |       |       |
| 34 | <i>type-name</i>                                       |            | R424          | R425  | R430  | R502  |       |
| 35 | <i>use-name</i>                                        |            | R522          | R1108 | R1109 |       |       |
| 36 | <i>variable-name</i>                                   |            | R544          | R547  | R549  | R601  | R610  |
| 37 |                                                        |            | R625          | R627  |       |       |       |
| 38 | <b>D.2.3 Terminal Symbols.</b>                         |            |               |       |       |       |       |
| 39 | %                                                      |            | R612          |       |       |       |       |
| 40 | '                                                      |            | R408          | R409  | R410  | R420  |       |
| 41 | (                                                      |            | R417          | R427  | R429  | R430  | R433  |
| 42 | _____                                                  |            | R502          | R503  | R504  | R505  | R506  |
| 43 | _____                                                  |            | R507          | R508  | R519  | R525  | R526  |
| 44 | _____                                                  |            | R527          | R528  | R535  | R538  | R541  |
| 45 |                                                        |            | R546          | R549  | R609  | R613  | R616  |
| 46 |                                                        |            | R622          | R624  | R626  | R628  | R701  |
| 47 |                                                        |            | R738          | R740  | R803  | R804  | R807  |
| 48 |                                                        |            | R809          | R813  | R821  | R837  | R839  |
| 49 |                                                        |            | R840          | R904  | R907  | R909  | R910  |
| 50 |                                                        |            | R916          | R919  | R920  | R921  | R923  |
| 51 |                                                        |            | R1002         | R1003 | R1206 | R1209 | R1210 |
| 52 |                                                        |            | R1216         | R1220 | R1223 | R1226 |       |
| 53 | /                                                      |            | R431          |       |       |       |       |



|    | Symbol  | Defined in | Referenced in |       |       |       |       |
|----|---------|------------|---------------|-------|-------|-------|-------|
| 1  | )       |            | R417          | R427  | R429  | R430  | R433  |
| 2  |         |            | R502          | R503  | R504  | R505  | R506  |
| 3  |         |            | R507          | R508  | R519  | R525  | R526  |
| 4  |         |            | R527          | R528  | R535  | R538  | R541  |
| 5  |         |            | R546          | R549  | R609  | R613  | R616  |
| 6  |         |            | R622          | R624  | R626  | R628  | R701  |
| 7  |         |            | R738          | R740  | R803  | R804  | R807  |
| 8  |         |            | R809          | R813  | R821  | R837  | R839  |
| 9  |         |            | R840          | R904  | R907  | R909  | R910  |
| 10 |         |            | R916          | R919  | R920  | R921  | R923  |
| 11 |         |            | R1002         | R1003 | R1206 | R1209 | R1210 |
| 12 |         |            | R1216         | R1220 | R1223 | R1226 |       |
| 13 | *       |            | R429          | R504  | R507  | R509  | R518  |
| 14 |         |            | R532          | R709  | R901  | R913  | R1214 |
| 15 |         |            | R1221         |       |       |       |       |
| 16 | **      |            | R708          |       |       |       |       |
| 17 | +       |            | R406          | R710  |       |       |       |
| 18 | ,       |            | R417          | R424  | R426  | R433  | R434  |
| 19 |         |            | R501          | R506  | R507  | R518  | R525  |
| 20 |         |            | R526          | R527  | R528  | R529  | R535  |
| 21 |         |            | R543          | R546  | R548  | R622  | R628  |
| 22 |         |            | R821          | R837  | R839  | R840  | R909  |
| 23 |         |            | R911          | R916  | R918  | R1107 |       |
| 24 | -       |            | R406          | R542  | R710  |       |       |
| 25 | .       |            | R414          | R704  | R724  | R1005 |       |
| 26 | .AND.   |            | R720          |       |       |       |       |
| 27 | .EQ.    |            | R714          |       |       |       |       |
| 28 | .EQV.   |            | R722          |       |       |       |       |
| 29 | .FALSE. |            | R421          |       |       |       |       |
| 30 | .GE.    |            | R714          |       |       |       |       |
| 31 | .GT.    |            | R714          |       |       |       |       |
| 32 | .LE.    |            | R714          |       |       |       |       |
| 33 | .LT.    |            | R714          |       |       |       |       |
| 34 | .NE.    |            | R714          |       |       |       |       |
| 35 | .NEQV.  |            | R722          |       |       |       |       |
| 36 | .NOT.   |            | R719          |       |       |       |       |
| 37 | .OR.    |            | R721          |       |       |       |       |
| 38 | .TRUE.  |            | R421          |       |       |       |       |
| 39 | /       |            | R524          | R530  | R543  | R548  | R709  |
| 40 |         |            | R1010         |       |       |       |       |
| 41 | /)      |            | R431          |       |       |       |       |
| 42 | //      |            | R712          |       |       |       |       |
| 43 | /=      |            | R714          |       |       |       |       |
| 44 | :       |            | R513          | R516  | R517  | R518  | R611  |
| 45 |         |            | R619          | R803  | R809  | R814  | R819  |
| 46 |         |            | R820          | R1010 | R1107 |       |       |
| 47 | ::      |            | R424          | R426  | R501  | R519  | R520  |
| 48 |         |            | R521          | R523  | R525  | R526  | R527  |
| 49 |         |            | R528          |       |       |       |       |
| 50 | <       |            | R714          |       |       |       |       |
| 51 | <=      |            | R714          |       |       |       |       |
| 52 | =       |            | R434          | R504  | R505  | R507  | R535  |
| 53 |         |            | R539          | R622  | R628  | R735  | R821  |
| 54 |         |            | R918          | R922  | R923  | R924  | R1206 |
| 55 |         |            | R1211         | R1226 |       |       |       |
| 56 | ==      |            | R714          |       |       |       |       |

# SYNTAX RULES

|    | Symbol      | Defined in | Referenced in                 |
|----|-------------|------------|-------------------------------|
| 1  | =>          |            | R736 R1108 R1109              |
| 2  | >           |            | R714                          |
| 3  | >=          |            | R714                          |
| 4  | A           |            | R411 R1005                    |
| 5  | ACCESS      |            | R924                          |
| 6  | ACCESS=     |            | R905                          |
| 7  | ACTION      |            | R924                          |
| 8  | ACTION=     |            | R905                          |
| 9  | ADVANCE=    |            | R912                          |
| 10 | ALLOCATABLE |            | R503 R526                     |
| 11 | ALLOCATE    |            | R622                          |
| 12 | ASSIGN      |            | R838                          |
| 13 | ASSIGNMENT  |            | R1206                         |
| 14 | B           |            | R408 R411 R1005               |
| 15 | BACKSPACE   |            | R919                          |
| 16 | BLANK       |            | R924                          |
| 17 | BLANK=      |            | R905                          |
| 18 | BLOCK       |            | R1111 R1112                   |
| 19 | BN          |            | R1015                         |
| 20 | BZ          |            | R1015                         |
| 21 | C           |            | R411                          |
| 22 | CALL        |            | R1210                         |
| 23 | CASE        |            | R809 R810                     |
| 24 | CHARACTER   |            | R502                          |
| 25 | CLOSE       |            | R907                          |
| 26 | COMMON      |            | R548                          |
| 27 | COMPLEX     |            | R502                          |
| 28 | CONTAINS    |            | R1225                         |
| 29 | CONTINUE    |            | R841                          |
| 30 | CYCLE       |            | R834                          |
| 31 | D           |            | R411 R415 R1005               |
| 32 | DATA        |            | R529 R1111 R1112              |
| 33 | DEALLOCATE  |            | R628                          |
| 34 | DEFAULT     |            | R813                          |
| 35 | DELIM       |            | R924                          |
| 36 | DELIM=      |            | R905                          |
| 37 | DIMENSION   |            | R427 R503 R525                |
| 38 | DIRECT      |            | R924                          |
| 39 | DO          |            | R819 R820 R825                |
| 40 | DOUBLE      |            | R502                          |
| 41 | E           |            | R411 R415 R1005               |
| 42 | ELSE        |            | R804 R805                     |
| 43 | ELSEWHERE   |            | R742                          |
| 44 | EN          |            | R1005                         |
| 45 | END         |            | R425 R743 R806 R811 R825      |
| 46 |             |            | R1103 R1106 R1112 R1203 R1218 |
| 47 |             |            | R1222                         |
| 48 | END=        |            | R912                          |
| 49 | ENDFILE     |            | R920                          |
| 50 | ENTRY       |            | R1223                         |
| 51 | EOR=        |            | R912                          |
| 52 | EQUIVALENCE |            | R545                          |
| 53 | ERR         |            | R922 R924                     |
| 54 | ERR=        |            | R905 R908 R912                |
| 55 | ES          |            | R1005                         |
| 56 | EXIST       |            | R924                          |

|    | Symbol    | Defined in | Referenced in |       |       |      |      |  |
|----|-----------|------------|---------------|-------|-------|------|------|--|
| 1  | EXIT      |            | R835          |       |       |      |      |  |
| 2  | EXTERNAL  |            | R503          | R1207 |       |      |      |  |
| 3  | F         |            | R411          | R1005 |       |      |      |  |
| 4  | FILE      |            | R924          |       |       |      |      |  |
| 5  | FILE=     |            | R905          |       |       |      |      |  |
| 6  | FMT=      |            | R912          |       |       |      |      |  |
| 7  | FORM      |            | R924          |       |       |      |      |  |
| 8  | FORM=     |            | R905          |       |       |      |      |  |
| 9  | FORMAT    |            | R1001         |       |       |      |      |  |
| 10 | FORMATTED |            | R924          |       |       |      |      |  |
| 11 | FUNCTION  |            | R1216         | R1218 |       |      |      |  |
| 12 | G         |            | R1005         |       |       |      |      |  |
| 13 | GO        |            | R836          | R837  | R839  |      |      |  |
| 14 | H         |            | R1016         |       |       |      |      |  |
| 15 | I         |            | R1005         |       |       |      |      |  |
| 16 | IF        |            | R803          | R804  | R806  | R807 | R840 |  |
| 17 | IMPLICIT  |            | R540          |       |       |      |      |  |
| 18 | IN        |            | R511          |       |       |      |      |  |
| 19 | INOUT     |            | R511          |       |       |      |      |  |
| 20 | INQUIRE   |            | R923          |       |       |      |      |  |
| 21 | INTEGER   |            | R502          |       |       |      |      |  |
| 22 | INTENT    |            | R503          | R519  |       |      |      |  |
| 23 | INTERFACE |            | R1202         | R1203 |       |      |      |  |
| 24 | INTRINSIC |            | R503          | R1208 |       |      |      |  |
| 25 | IOLength  |            | R923          |       |       |      |      |  |
| 26 | IOSTAT    |            | R922          | R924  |       |      |      |  |
| 27 | IOSTAT=   |            | R905          | R908  | R912  |      |      |  |
| 28 | KIND      |            | R505          |       |       |      |      |  |
| 29 | KIND=     |            | R506          |       |       |      |      |  |
| 30 | L         |            | R1005         |       |       |      |      |  |
| 31 | LEN       |            | R507          |       |       |      |      |  |
| 32 | LEN=      |            | R506          |       |       |      |      |  |
| 33 | LOGICAL   |            | R502          |       |       |      |      |  |
| 34 | MODULE    |            | R1105         | R1106 | R1205 |      |      |  |
| 35 | NAME      |            | R924          |       |       |      |      |  |
| 36 | NAMED     |            | R924          |       |       |      |      |  |
| 37 | NAMelist  |            | R543          |       |       |      |      |  |
| 38 | NEXTREC   |            | R924          |       |       |      |      |  |
| 39 | NML=      |            | R912          |       |       |      |      |  |
| 40 | NONE      |            | R540          |       |       |      |      |  |
| 41 | NULLIFY   |            | R626          |       |       |      |      |  |
| 42 | NUMBER    |            | R924          |       |       |      |      |  |
| 43 | O         |            | R409          | R1005 |       |      |      |  |
| 44 | ONLY      |            | R1107         |       |       |      |      |  |
| 45 | OPEN      |            | R904          |       |       |      |      |  |
| 46 | OPENED    |            | R924          |       |       |      |      |  |
| 47 | OPERATOR  |            | R1206         |       |       |      |      |  |
| 48 | OPTIONAL  |            | R503          | R520  |       |      |      |  |
| 49 | OUT       |            | R511          |       |       |      |      |  |
| 50 | P         |            | R1010         |       |       |      |      |  |
| 51 | PAD       |            | R924          |       |       |      |      |  |
| 52 | PAD=      |            | R905          |       |       |      |      |  |
| 53 | PARAMETER |            | R503          | R538  |       |      |      |  |
| 54 | PAUSE     |            | R844          |       |       |      |      |  |
| 55 | POINTER   |            | R427          | R503  | R527  |      |      |  |
| 56 | POSITION  |            | R924          |       |       |      |      |  |

**SYNTAX RULES**

|    | Symbol      | Defined in | Referenced in |       |      |           |
|----|-------------|------------|---------------|-------|------|-----------|
| 1  | POSITION=   |            | R905          |       |      |           |
| 2  | PRECISION   |            | R502          |       |      |           |
| 3  | PRINT       |            | R911          |       |      |           |
| 4  | PRIVATE     |            | R423          | R510  |      |           |
| 5  | PROCEDURE   |            | R1205         |       |      |           |
| 6  | PROGRAM     |            | R1102         | R1103 |      |           |
| 7  | PUBLIC      |            | R510          |       |      |           |
| 8  | READ        |            | R909          | R924  |      |           |
| 9  | READWRITE   |            | R924          |       |      |           |
| 10 | REAL        |            | R502          |       |      |           |
| 11 | REC=        |            | R912          |       |      |           |
| 12 | RECL        |            | R924          |       |      |           |
| 13 | RECL=       |            | R905          |       |      |           |
| 14 | RECURSIVE   |            | R1217         | R1220 |      |           |
| 15 | RESULT      |            | R1216         | R1223 |      |           |
| 16 | RETURN      |            | R1224         |       |      |           |
| 17 | REWIND      |            | R921          |       |      |           |
| 18 | S           |            | R1014         |       |      |           |
| 19 | SAVE        |            | R503          | R523  |      |           |
| 20 | SELECT      |            | R809          | R811  |      |           |
| 21 | SEQUENCE    |            | R423          |       |      |           |
| 22 | SEQUENTIAL  |            | R924          |       |      |           |
| 23 | SIZE=       |            | R912          |       |      |           |
| 24 | SP          |            | R1014         |       |      |           |
| 25 | SS          |            | R1014         |       |      |           |
| 26 | STAT        |            | R622          | R628  |      |           |
| 27 | STATUS=     |            | R905          | R908  |      |           |
| 28 | STOP        |            | R842          |       |      |           |
| 29 | SUBROUTINE  |            | R1220         | R1222 |      |           |
| 30 | T           |            | R1012         |       |      |           |
| 31 | TARGET      |            | R503          | R528  |      |           |
| 32 | THEN        |            | R803          | R804  |      |           |
| 33 | TL          |            | R1012         |       |      |           |
| 34 | TO          |            | R836          | R837  | R838 | R839      |
| 35 | TR          |            | R1012         |       |      |           |
| 36 | TYPE        |            | R424          | R425  | R502 |           |
| 37 | UNFORMATTED |            | R924          |       |      |           |
| 38 | UNIT        |            | R922          | R924  |      |           |
| 39 | UNIT=       |            | R905          | R908  | R912 |           |
| 40 | USE         |            | R1107         |       |      |           |
| 41 | WHERE       |            | R738          | R740  | R743 |           |
| 42 | WHILE       |            | R821          |       |      |           |
| 43 | WRITE       |            | R910          | R924  |      |           |
| 44 | X           |            | R1012         |       |      |           |
| 45 | Z           |            | R410          | R1005 |      |           |
| 46 | -           |            | R303          | R404  | R413 | R420 R421 |

# ANNEX E. PERMUTED INDEX FOR HEADINGS

(This Annex is not part of this standard, but is included for information only.)

|                                   |                                   |
|-----------------------------------|-----------------------------------|
| 11.3.3.7 Data                     | Abstraction                       |
| 9.2.1.2 File                      | Access                            |
| 9.2.1.2.1 Sequential              | Access                            |
| 9.2.1.2.2 Direct                  | Access                            |
| Statement 9.6.1.7                 | ACCESS= Specifier in the INQUIRE  |
| Statement 9.3.4.3                 | ACCESS= Specifier in the OPEN     |
| 5.1.2.2                           | Accessibility Attribute           |
| 5.2.3                             | Accessibility Statements          |
| Statement 9.6.1.17                | ACTION= Specifier in the INQUIRE  |
| Statement 9.3.4.8                 | ACTION= Specifier in the OPEN     |
| 8.1.4.3                           | Active and Inactive DO Constructs |
| 12.4.1                            | Actual Argument List              |
| 9.4.1.8                           | Advance Specifier                 |
| Input/Output 9.2.1.3.1            | Advancing and Nonadvancing        |
| 11.3.3.4 Global                   | Allocatable Arrays                |
| 6.3.1.1 Allocation of             | Allocatable Arrays                |
| 6.3.3.1 Deallocation of           | Allocatable Arrays                |
| 5.1.2.9                           | ALLOCATABLE Attribute             |
| 5.2.6                             | ALLOCATABLE Statement             |
| 6.3.1                             | ALLOCATE Statement                |
| 6.3.1.1                           | Allocation of Allocatable Arrays  |
| 6.3.1.2                           | Allocation of Pointer Targets     |
| 14.8                              | Allocation Status                 |
| /Arguments Associated with        | Alternate Return Indicators       |
| 14.7.2 Variables That Are         | Always Defined                    |
| 14.6.1.1                          | Argument Association              |
| 13.3 Positional Arguments or      | Argument Keywords                 |
| 14.1.2.6                          | Argument Keywords                 |
| 12.4.1 Actual                     | Argument List                     |
| Function 13.10.1                  | Argument Presence Inquiry         |
| Function 13.4                     | Argument Presence Inquiry         |
| 12.2.1 Characteristics of Dummy   | Arguments                         |
| Characteristics of Asterisk Dummy | Arguments 12.2.1.3                |
| on Entities Associated with Dummy | Arguments 12.5.2.9 Restrictions   |
| Elemental Intrinsic Subroutine    | Arguments 13.2.2                  |
| 13.8.1 The Shape of Array         | Arguments                         |
| 13.8.2 Mask                       | Arguments                         |
| Elemental Intrinsic Function      | Arguments and Results 13.2.1      |
| Alternate Return/ 12.4.1.3        | Arguments Associated with         |
| Data Objects 12.4.1.1             | Arguments Associated with Dummy   |
| Procedures 12.4.1.2               | Arguments Associated with Dummy   |
| 12.5.2.8 Restrictions on Dummy    | Arguments Not Present             |
| 13.3 Positional                   | Arguments or Argument Keywords    |
| 8.2.5                             | Arithmetic IF Statement           |
| 2.4.5                             | Array                             |
| 5.1.2.4.1 Explicit-Shape          | Array                             |
| 5.1.2.4.2 Assumed-Shape           | Array                             |
| 5.1.2.4.3 Deferred-Shape          | Array                             |
| 5.1.2.4.4 Assumed-Size            | Array                             |
| 13.8.1 The Shape of               | Array Arguments                   |
| General Form of the Masked        | Array Assignment 7.5.3.1          |
| 7.5.3 Masked                      | Array Assignment WHERE            |
| 7.5.3.2 Interpretation of Masked  | Array Assignments                 |
| 13.10.16                          | Array Construction Functions      |
| 13.8.6                            | Array Construction Functions      |
| 5.5.1.3 Array Names and           | Array Element Designators         |
| 6.2.2.2                           | Array Element Order               |
| 6.2.2.1                           | Array Elements                    |
| 6.2.2                             | Array Elements and Array Sections |
| 7.1.6 Scalar and                  | Array Expressions                 |
| 13.10.15                          | Array Inquiry Functions           |
| 13.8.5                            | Array Inquiry Functions           |
| 13.8                              | Array Intrinsic Functions         |
| 13.10.19                          | Array Location Functions          |
| 13.8.9                            | Array Location Functions          |
| 13.10.18                          | Array Manipulation Functions      |
| 13.8.8                            | Array Manipulation Functions      |
| Designators 5.5.1.3               | Array Names and Array Element     |
| 13.10.14                          | Array Reduction Functions         |
| 13.8.4                            | Array Reduction Functions         |
| 13.10.17                          | Array Reshape Function            |
| 13.8.7                            | Array Reshape Function            |
| 6.2.2 Array Elements and          | Array Sections                    |
| 6.2.2.3                           | Array Sections                    |
| 4.5 Construction of               | Array Values                      |
| 11.3.3.4 Global Allocatable       | Arrays                            |
| 6.2                               | Arrays                            |
| 6.2.1 Whole                       | Arrays                            |

# Permuted Index for Headings

|                                   |                                    |
|-----------------------------------|------------------------------------|
| 6.3.1.1 Allocation of Allocatable | Arrays                             |
| Deallocation of Allocatable       | Arrays 6.3.3.1                     |
| Statement 8.2.4                   | ASSIGN and Assigned GO TO          |
| 8.2.4 ASSIGN and                  | Assigned GO TO Statement           |
| 4.4.5 Derived-Type Operations and | Assignment                         |
| 7 Expressions and                 | Assignment                         |
| 7.5                               | Assignment                         |
| 7.5.2 Pointer                     | Assignment                         |
| General Form of the Masked Array  | Assignment 7.5.3.1                 |
| 7.5.1.4 Intrinsic                 | Assignment Conformance Rules       |
| 7.5.1                             | Assignment Statement               |
| 7.5.1.2 Intrinsic                 | Assignment Statement               |
| 7.5.1.3 Defined                   | Assignment Statement               |
| 7.5.1.6 Interpretation of Defined | Assignment Statements              |
| 14.5 Scope of the                 | Assignment Symbol                  |
| 7.5.3 Masked Array                | Assignment WHERE                   |
| 12.3.2.1.2 Defined                | Assignments                        |
| Interpretation of Intrinsic       | Assignments 7.5.1.5                |
| Interpretation of Masked Array    | Assignments 7.5.3.2                |
| Indicators 12.4.1.3 Arguments     | Associated with Alternate Return   |
| 12.5.2.9 Restrictions on Entities | Associated with Dummy Arguments    |
| Objects 12.4.1.1 Arguments        | Associated with Dummy Data         |
| 12.4.1.2 Arguments                | Associated with Dummy Procedures   |
| 11.3.2 The USE Statement and Use  | Association                        |
| 12.1.2.2.1 Host                   | Association                        |
| Host Association and USE          | Association 12.1.2.2.2             |
| 12.4.1.4 Sequence                 | Association                        |
| 14.6                              | Association                        |
| 14.6.1 Name                       | Association                        |
| 14.6.1.1 Argument                 | Association                        |
| 14.6.1.2 Use Association and Host | Association                        |
| 14.6.2 Pointer                    | Association                        |
| 14.6.3 Storage                    | Association                        |
| 2.5.6                             | Association                        |
| 5.5.1.1 Equivalence               | Association                        |
| 5.5.2.3 Common                    | Association                        |
| 6.3 Dynamic                       | Association                        |
| 14 Scope,                         | Association, and Definition        |
| 14.6.1.2 Use                      | Association and Host Association   |
| 12.1.2.2.2 Host                   | Association and USE Association    |
| 5.5 Storage                       | Association of Data Objects        |
| Objects 14.6.3.3                  | Association of Scalar Data         |
| 14.6.3.2                          | Association of Storage Sequences   |
| 14.6.2.1 Pointer                  | Association Status                 |
| 14.6.2.3 Relationship Between     | Association Status and Definition/ |
| Function 13.10.20 Pointer         | Association Status Inquiry         |
| Functions 13.8.10 Pointer         | Association Status Inquiry         |
| 1.5.2                             | Assumed Syntax Rules               |
| 5.1.2.4.2                         | Assumed-Shape Array                |
| 5.1.2.4.4                         | Assumed-Size Array                 |
| 12.2.1.3 Characteristics of       | Asterisk Dummy Arguments           |
| 5.1.2.1 PARAMETER                 | Attribute                          |
| 5.1.2.10 EXTERNAL                 | Attribute                          |
| 5.1.2.11 INTRINSIC                | Attribute                          |
| 5.1.2.2 Accessibility             | Attribute                          |
| 5.1.2.3 INTENT                    | Attribute                          |
| 5.1.2.4 DIMENSION                 | Attribute                          |
| 5.1.2.5 SAVE                      | Attribute                          |
| 5.1.2.6 OPTIONAL                  | Attribute                          |
| 5.1.2.7 POINTER                   | Attribute                          |
| 5.1.2.8 TARGET                    | Attribute                          |
| 5.1.2.9 ALLOCATABLE               | Attribute                          |
| 12.5.2.1 Effects of INTENT        | Attribute on Subprograms           |
| Statements 5.2                    | Attribute Specification            |
| 5.1.2                             | Attributes                         |
| 9.5.1                             | BACKSPACE Statement                |
| Events That Cause Variables to    | Become Defined 14.7.5              |
| Events That Cause Variables to    | Become Undefined 14.7.6            |
| 7.3.2                             | Binary Defined Operation           |
| 13.9.3                            | Bit Copy Subroutine                |
| 13.10.9                           | Bit Inquiry Function               |
| Procedures 13.5.7                 | Bit Manipulation and Inquiry       |
| 13.10.10                          | Bit Manipulation Functions         |
| Character, Kind, Logical, and     | Bit Procedures /Mathematical,      |
| between Named Common and          | Blank Common 5.5.2.4 Differences   |
| Statement 9.6.1.15                | BLANK= Specifier in the INQUIRE    |
| Statement 9.3.4.6                 | BLANK= Specifier in the OPEN       |
| 10.9.1.5                          | Blanks                             |
| 12.3.2.1 Procedure Interface      | Block                              |
| 2.2.3.4 Procedure Interface       | Block                              |
| 5.5.2.2 Size of a Common          | Block                              |

|                                   |                                    |
|-----------------------------------|------------------------------------|
| 8.1.1.3 Execution of a            | Block                              |
| 11.4                              | Block Data Program Units           |
| 8.1.4.1.1 Form of the             | Block DO Construct                 |
| 5.5.2.1 Common                    | Block Storage Sequence             |
| 11.3.3.1 Identical Common         | Blocks                             |
| 14.1.2.1 Common                   | Blocks                             |
| Executable Constructs Containing  | Blocks 8.1                         |
| 8.1.1 Rules Governing             | Blocks                             |
| 8.1.1.1 Executable Constructs in  | Blocks                             |
| 8.1.1.2 Control Flow in           | Blocks                             |
| 10.6.6                            | BN and BZ Editing                  |
| 9.4.1.5 Error                     | Branch                             |
| 9.4.1.6 End-of-File               | Branch                             |
| 9.4.1.7 End-of-Record             | Branch                             |
| 8.2                               | Branching                          |
| 10.6.6 BN and                     | BZ Editing                         |
| 8.1.3                             | CASE Construct                     |
| 8.1.3.1 Form of the               | CASE Construct                     |
| 8.1.3.2 Execution of a            | CASE Construct                     |
| 8.1.3.3 Examples of               | CASE Constructs                    |
| 14.7.5 Events That                | Cause Variables to Become Defined  |
| Undefined 14.7.6 Events That      | Cause Variables to Become          |
| 5.1.1.5                           | CHARACTER                          |
| Descriptor 10.7.1                 | Character Constant Edit            |
| 3.3.1.3.2                         | Character Context Continuation     |
| 9.4.1.9                           | Character Count                    |
| 10.5.3                            | Character Editing                  |
| 10.5.4.3 Generalized              | Character Editing                  |
| 10.1.2                            | Character Format Specification     |
| 13.10.4                           | Character Functions                |
| 13.5.3                            | Character Functions                |
| 13.10.5                           | Character Inquiry Function         |
| 13.5.4                            | Character Inquiry Function         |
| 7.1.7.4 Evaluation of the         | Character Intrinsic Operation      |
| 7.2.2                             | Character Intrinsic Operation      |
| 13.5 Numeric, Mathematical,       | Character, Kind, Logical, and Bit/ |
| 5.5.1.2 Equivalence of            | Character Objects                  |
| 3.1 Processor                     | Character Set                      |
| 10.7                              | Character String Edit Descriptors  |
| 4.3.2.1                           | Character Type                     |
| 1.5.3 Syntax Conventions and      | Characteristics                    |
| Arguments 12.2.1.3                | Characteristics of Asterisk Dummy  |
| Arguments 12.2.1                  | Characteristics of Dummy           |
| Objects 12.2.1.1                  | Characteristics of Dummy Data      |
| Procedures 12.2.1.2               | Characteristics of Dummy           |
| Results 12.2.2                    | Characteristics of Function        |
| 12.2                              | Characteristics of Procedures      |
| 3.1.4 Special                     | Characters                         |
| 3.1.5 Other                       | Characters                         |
| Source Form 3                     | Characters, Lexical Tokens, and    |
| Definition 12.1.2 Procedure       | Classification by Means of         |
| 12.1.1 Procedure                  | Classification by Reference        |
| 12.1 Procedure                    | Classifications                    |
| 9.3.5 The                         | CLOSE Statement                    |
| 9.3.5.1 STATUS= Specifier in the  | CLOSE Statement                    |
| 4.3.2.1.1                         | Collating Sequence                 |
| 10.6.3                            | Colon Editing                      |
| 3.3.1.1 Free Form                 | Commentary                         |
| 3.3.2.1 Fixed Form                | Commentary                         |
| between Named Common and Blank    | Common 5.5.2.4 Differences         |
| 5.5.2.4 Differences between Named | Common and Blank Common            |
| 5.5.2.5 Restrictions on           | Common and Equivalence             |
| 5.5.2.3                           | Common Association                 |
| 5.5.2.2 Size of a                 | Common Block                       |
| 5.5.2.1                           | Common Block Storage Sequence      |
| 11.3.3.1 Identical                | Common Blocks                      |
| 14.1.2.1                          | Common Blocks                      |
| 5.5.2                             | COMMON Statement                   |
| 1.4.1 FORTRAN 77                  | Compatibility                      |
| 5.1.1.4                           | COMPLEX                            |
| 10.5.1.2 Real and                 | Complex Editing                    |
| 10.5.1.2.5                        | Complex Editing                    |
| 10.5.4.1.2 Generalized Real and   | Complex Editing                    |
| 7.2.1.2                           | Complex Exponentiation             |
| 4.3.1.3                           | Complex Type                       |
| 14.1.2.5                          | Components                         |
| 6.1.2 Structure                   | Components                         |
| 8.2.3                             | Computed GO TO Statement           |
| 4.1 The                           | Concept of Data Type               |
| 2 Fortran Terms and               | Concepts                           |
| 2.2 Program Unit                  | Concepts                           |

# Permuted Index for Headings

|                                   |                                 |
|-----------------------------------|---------------------------------|
| 2.3 Execution                     | Concepts                        |
| 2.4 Data                          | Concepts                        |
| End-of-Record, and End-of-File    | Conditions 9.4.3 Error,         |
| Intrinsic Operations 7.1.5        | Conformability Rules for        |
| 1.4                               | Conformance                     |
| 7.5.1.4 Intrinsic Assignment      | Conformance Rules               |
| 9.3 File                          | Connection                      |
| 9.3.2                             | Connection of a File to a Unit  |
| 2.4.3.1.2                         | Constant                        |
| 10.7.1 Character                  | Constant Edit Descriptor        |
| 7.1.6.1                           | Constant Expression             |
| 2.4.3.1.3                         | Constant Subobject              |
| 3.2.3                             | Constants                       |
| 4.1.2                             | Constants                       |
| 8.1.2 IF                          | Construct                       |
| 8.1.2.1 Form of the IF            | Construct                       |
| 8.1.2.2 Execution of an IF        | Construct                       |
| 8.1.3 CASE                        | Construct                       |
| 8.1.3.1 Form of the CASE          | Construct                       |
| 8.1.3.2 Execution of a CASE       | Construct                       |
| 8.1.4 DO                          | Construct                       |
| 8.1.4.1 Forms of the DO           | Construct                       |
| 8.1.4.1.1 Form of the Block DO    | Construct                       |
| 8.1.4.1.2 Form of the Nonblock DO | Construct                       |
| 8.1.4.2 Range of the DO           | Construct                       |
| 8.1.4.4 Execution of a DO         | Construct                       |
| 13.10.16 Array                    | Construction Functions          |
| 13.8.6 Array                      | Construction Functions          |
| 4.5                               | Construction of Array Values    |
| Values 4.4.4                      | Construction of Derived-Type    |
| 8.1.2.3 Examples of IF            | Constructs                      |
| 8.1.3.3 Examples of CASE          | Constructs                      |
| 8.1.4.3 Active and Inactive DO    | Constructs                      |
| 8.1.4.5 Examples of DO            | Constructs                      |
| 8.1 Executable                    | Constructs Containing Blocks    |
| 8.1.1.1 Executable                | Constructs in Blocks            |
| 8.1 Executable Constructs         | Containing Blocks               |
| 12.5.2.7                          | CONTAINS Statement              |
| 3.3.1.3.1 Noncharacter            | Context Continuation            |
| 3.3.1.3.2 Character               | Context Continuation            |
| 3.3.1.3 Free Form Statement       | Continuation                    |
| 3.3.1.3.1 Noncharacter Context    | Continuation                    |
| 3.3.1.3.2 Character Context       | Continuation                    |
| 3.3.2.3 Fixed Form Statement      | Continuation                    |
| 8.3                               | CONTINUE Statement              |
| 10.4 Positioning by Format        | Control                         |
| 8 Execution                       | Control                         |
| 10.6                              | Control Edit Descriptors        |
| 8.1.1.2                           | Control Flow in Blocks          |
| 9.4.1                             | Control Information List        |
| 1.5.4 Text                        | Conventions                     |
| 1.5.3 Syntax                      | Conventions and Characteristics |
| 13.9.3 Bit                        | Copy Subroutine                 |
| 9.4.1.9 Character                 | Count                           |
| 8.1.4.4.2 The Execution           | Cycle                           |
| 8.1.4.4.3                         | CYCLE Statement                 |
| 10.5.1.2.2 E and                  | D Editing                       |
| 11.3.3.2 Global                   | Data                            |
| Models for Integer and Real       | Data 13.7.1                     |
| 11.3.3.7                          | Data Abstraction                |
| 2.4                               | Data Concepts                   |
| 10.5                              | Data Edit Descriptors           |
| 2.4.3                             | Data Entity                     |
| 2.4.3.1                           | Data Object                     |
| Specifications 5                  | Data Object Declarations and    |
| 12.2.1.1 Characteristics of Dummy | Data Objects                    |
| Arguments Associated with Dummy   | Data Objects 12.4.1.1           |
| 14.6.3.3 Association of Scalar    | Data Objects                    |
| 5.5 Storage Association of        | Data Objects                    |
| 6 Use of                          | Data Objects                    |
| 11.4 Block                        | Data Program Units              |
| 5.2.9                             | DATA Statement                  |
| 11.3.3.3                          | Data Structures                 |
| 9.2.1.3.2 File Position Prior to  | Data Transfer                   |
| 9.2.1.3.3 File Position after     | Data Transfer                   |
| 9.4.4.1 Direction of              | Data Transfer                   |
| 9.4.4.4                           | Data Transfer                   |
| 9.4.4.4.1 Unformatted             | Data Transfer                   |
| 9.4.4.4.2 Formatted               | Data Transfer                   |
| 9.4.2                             | Data Transfer Input/Output List |
| Statement 9.4.4 Execution of a    | Data Transfer Input/Output      |



|                                  |                                                 |
|----------------------------------|-------------------------------------------------|
| 9.4                              | Data Transfer Statements                        |
| 9.4.6                            | Termination of Data Transfer Statements         |
| 2.4.1                            | Data Type                                       |
| 4.1                              | The Concept of Data Type                        |
| Shape of a Primary 7.1.4.1       | Data Type, Type Parameters, and                 |
| Shape of an Expression 7.1.4     | Data Type, Type Parameters, and                 |
| Shape of the Result of / 7.1.4.2 | Data Type, Type Parameters, and                 |
| 4                                | Intrinsic and Derived Data Types                |
| 4.3                              | Intrinsic Data Types                            |
| 2.4.2                            | Data Value                                      |
| 13.9.1                           | Date and Time Subroutines                       |
| 6.3.3                            | DEALLOCATE Statement                            |
| Arrays 6.3.3.1                   | Deallocation of Allocatable                     |
| 6.3.3.2                          | Deallocation of Pointer Targets                 |
| 2.5.3                            | Declaration                                     |
| 5.1                              | Type Declaration Statements                     |
| 5                                | Data Object Declarations and Specifications     |
| 5.1.2.4.3                        | Deferred-Shape Array                            |
| 14.7.2                           | Variables That Are Always Defined               |
| Variables That Are Initially     | Defined 14.7.3                                  |
| That Cause Variables to Become   | Defined 14.7.5 Events                           |
| 7.5.1.3                          | Defined Assignment Statement                    |
| 7.5.1.6                          | Interpretation of Defined Assignment Statements |
| 12.3.2.1.2                       | Defined Assignments                             |
| 12.5.2                           | Procedures Defined by Subprograms               |
| 7.1.7.7                          | Evaluation of a Defined Operation               |
| 7.3.1                            | Unary Defined Operation                         |
| 7.3.2                            | Binary Defined Operation                        |
| 12.3.2.1.1                       | Defined Operations                              |
| 7.1.3                            | Defined Operations                              |
| 7.3                              | Interpretation of Defined Operations            |
| Classification by Means of       | Definition 12.1.2 Procedure                     |
| 12.5                             | Procedure Definition                            |
| 12.5.1                           | Intrinsic Procedure Definition                  |
| 14                               | Scope, Association, and Definition              |
| 2.5.4                            | Definition                                      |
| 4.4.1                            | Derived-Type Definition                         |
| Variables 14.7                   | Definition and Undefined of                     |
| Subobjects 14.7.1                | Definition of Objects and                       |
| Other Than Fortran 12.5.3        | Definition of Procedures by Means               |
| 14.6.2.2                         | Pointer Definition Status                       |
| Between Association Status and   | Definition Status /Relationship                 |
| 1.6                              | Deleted and Obsolescent Features                |
| 1.6.1                            | Nature of Deleted Features                      |
| Statement 9.6.1.21               | DELIM= Specifier in the INQUIRE                 |
| Statement 9.3.4.9                | DELIM= Specifier in the OPEN                    |
| 3.2.6                            | Delimiters                                      |
| 4                                | Intrinsic and Derived Data Types                |
| 2.4.1.2                          | Derived Type                                    |
| 5.1.1.7                          | Derived Type                                    |
| 4.4                              | Derived Types                                   |
| 4.4.2                            | Determination of Derived Types                  |
| 4.4.1                            | Derived-Type Definition                         |
| Assignment 4.4.5                 | Derived-Type Operations and                     |
| 4.4.3                            | Derived-Type Values                             |
| 4.4.4                            | Construction of Derived-Type Values             |
| 10.7.1                           | Character Constant Edit Descriptor              |
| 10.2.1                           | Edit Descriptors                                |
| 10.5                             | Data Edit Descriptors                           |
| 10.6                             | Control Edit Descriptors                        |
| 10.7                             | Character String Edit Descriptors               |
| 2.5.1                            | Name and Designator                             |
| Array Names and Array Element    | Designators 5.5.1.3                             |
| 4.4.2                            | Determination of Derived Types                  |
| and Blank Common 5.5.2.4         | Differences between Named Common                |
| 3.1.2                            | Digits                                          |
| 5.1.2.4                          | DIMENSION Attribute                             |
| 5.2.5                            | DIMENSION Statement                             |
| 9.2.1.2.2                        | Direct Access                                   |
| Statement 9.6.1.9                | DIRECT= Specifier in the INQUIRE                |
| 9.4.4.1                          | Direction of Data Transfer                      |
| 7.2.1.1                          | Integer Division                                |
| 5.1.1.3                          | DOUBLE PRECISION                                |
| 12.2.1                           | Characteristics of Dummy Arguments              |
| Characteristics of Asterisk      | Dummy Arguments 12.2.1.3                        |
| on Entities Associated with      | Dummy Arguments /Restrictions                   |
| 12.5.2.8                         | Restrictions on Dummy Arguments Not Present     |
| 12.2.1.1                         | Characteristics of Dummy Data Objects           |
| Arguments Associated with        | Dummy Data Objects 12.4.1.1                     |
| 12.1.2.3                         | Dummy Procedures                                |
| 12.2.1.2                         | Characteristics of Dummy Procedures             |

# Permuted Index for Headings

Arguments Associated with Dummy Procedures 12.4.1.2  
     6.3 Dynamic Association  
     10.5.1.2.2 E and D Editing  
 10.7.1 Character Constant Edit Descriptor  
     10.2.1 Edit Descriptors  
     10.5 Data Edit Descriptors  
     10.6 Control Edit Descriptors  
 10.7 Character String Edit Descriptors  
     10 Input/Output Editing  
     10.5.1 Numeric Editing  
     10.5.1.1 Integer Editing  
 10.5.1.2 Real and Complex Editing  
     10.5.1.2.1 F Editing  
     10.5.1.2.2 E and D Editing  
     10.5.1.2.3 EN Editing  
     10.5.1.2.4 ES Editing  
     10.5.1.2.5 Complex Editing  
     10.5.2 Logical Editing  
     10.5.3 Character Editing  
     10.5.4 Generalized Editing  
     10.5.4.1 Generalized Numeric Editing  
     10.5.4.1.1 Generalized Integer Editing  
 Generalized Real and Complex Editing 10.5.4.1.2  
     10.5.4.2 Generalized Logical Editing  
 10.5.4.3 Generalized Character Editing  
     10.6.1 Position Editing  
     10.6.1.1 T, TL, and TR Editing  
     10.6.1.2 X Editing  
     10.6.2 Slash Editing  
     10.6.3 Colon Editing  
     10.6.4 S, SP, and SS Editing  
     10.6.5 P Editing  
     10.6.6 BN and BZ Editing  
     10.7.2 H Editing  
     10.9.2.1 Namelist Output Editing  
 Subprograms 12.5.2.1 Effects of INTENT Attribute on  
 5.5.1.3 Array Names and Array Element Designators  
     6.2.2.2 Array Element Order  
 Arguments and Results 13.2.1 Elemental Intrinsic Function  
     Reference 12.4.3 Elemental Intrinsic Function  
     13.2 Elemental Intrinsic Procedures  
     Arguments 13.2.2 Elemental Intrinsic Subroutine  
     Reference 12.4.5 Elemental Intrinsic Subroutine  
     6.2.2.1 Array Elements  
     6.2.2 Array Elements and Array Sections  
     10.5.1.2.3 EN Editing  
     2.3.3 The END Statement  
     9.1.3 Endfile Record  
     9.5.2 ENDFILE Statement  
     9.4.1.6 End-of-File Branch  
 9.4.3 Error, End-of-Record, and End-of-File Conditions  
     9.4.3 Error, End-of-Record, and End-of-File  
     9.4.1.7 End-of-Record Branch  
     14.1.1 Global Entities  
     14.1.2 Local Entities  
     14.1.3 Statement Entities  
     12.5.2.9 Restrictions on Entities Associated with Dummy/  
     11.3.3.8 Public Entities Renamed  
     2.4.3 Data Entity  
     12.5.2.5 ENTRY Statement  
 Restrictions on Common and Equivalence 5.5.2.5  
     5.5.1.1 Equivalence Association  
     5.5.1.2 Equivalence of Character Objects  
     5.5.1 EQUIVALENCE Statement  
     5.5.1.4 Restrictions on EQUIVALENCE Statements  
     9.4.1.5 Error Branch  
 End-of-File Conditions 9.4.3 Error, End-of-Record, and  
     10.5.1.2.4 ES Editing  
 Procedure References to Names Not Established 14.1.2.4.3 Resolving  
     /Procedure References to Names Established to be Generic  
     /Procedure References to Names Established to be Only Specific  
     9.4.4.3 Establishing a Format  
     7.1.7.7 Evaluation of a Defined Operation  
     Operations 7.1.7.6 Evaluation of Logical Intrinsic  
     Operations 7.1.7.3 Evaluation of Numeric Intrinsic  
     7.1.7.1 Evaluation of Operands  
     7.1.7 Evaluation of Operations  
     Intrinsic Operations 7.1.7.5 Evaluation of Relational  
     Intrinsic Operation 7.1.7.4 Evaluation of the Character  
     Become Defined 14.7.5 Events That Cause Variables to  
     Become Undefined 14.7.6 Events That Cause Variables to

- 10.8.1.2 List-Directed Input Example
- 10.9.1.6 Namelist Input Example
  - 8.1.3.3 Examples of CASE Constructs
  - 8.1.4.5 Examples of DO Constructs
  - 8.1.2.3 Examples of IF Constructs
  - 11.3.3 Examples of the Use of Modules
- 1.3 Inclusions and Exclusions
  - 1.3.2 Exclusions
    - Blocks 8.1 Executable Constructs Containing
      - 8.1.1.1 Executable Constructs in Blocks
- 11.1.2 Main Program Executable Part
  - 2.2.1 Executable Program
- Statements 2.3.1 Executable/Nonexecutable
- Statement 9.6.1.2 EXIST= Specifier in the INQUIRE
  - 9.2.1.1 File Existence
  - 9.3.1 Unit Existence
- Methods 10.1 Explicit Format Specification
  - 12.3.1.1 Explicit Interface
- 12.3.1 Implicit and Explicit Interfaces
  - 5.1.2.4.1 Explicit-Shape Array
- 7.2.1.2 Complex Exponentiation
  - 7.1 Expressions
    - 7.1.1.2 Level-1 Expressions
    - 7.1.1.3 Level-2 Expressions
    - 7.1.1.4 Level-3 Expressions
    - 7.1.1.5 Level-4 Expressions
    - 7.1.1.6 Level-5 Expressions
- 7.1.6 Scalar and Array Expressions
  - 7 Expressions and Assignment
- 11.3.3.6 Operator Extensions
  - 5.1.2.10 EXTERNAL Attribute
  - 9.2.1 External Files
    - 14.3 Scope of External Input/Output Units
- Procedures 12.1.2.2 External, Internal, and Module
  - 2.2.3.1 External Procedure
  - 12.3.2.2 EXTERNAL Statement
    - 11.2 External Subprograms
  - 10.5.1.2.1 F Editing
  - 10.6.5.1 Scale Factor
- 1.6 Deleted and Obsolescent Features
  - 1.6.1 Nature of Deleted Features
  - 1.6.2 Nature of Obsolescent Features
    - 10.2.2 Fields
      - 9.2.1.2 File Access
        - 9.3 File Connection
          - 9.2.1.1 File Existence
            - 9.6 File Inquiry
              - 9.2.1.3 File Position
                - 9.2.1.3.3 File Position after Data Transfer
  - Transfer 9.2.1.3.2 File Position Prior to Data
    - 9.5 File Positioning Statements
      - 9.2.2.1 Internal File Properties
      - 9.2.2.2 Internal File Restrictions
- Statement 9.6.1.1 FILE= Specifier in the INQUIRE
- Statement 9.3.4.1 FILE= Specifier in the OPEN
- 9.3.2 Connection of a File to a Unit
  - 9.2 Files
    - 9.2.1 External Files
      - 9.2.2 Internal Files
        - 3.3.2.1 Fixed Form Commentary
        - 3.3.2.3 Fixed Form Statement Continuation
        - 3.3.2.2 Fixed Form Statement Separation
        - 3.3.2.4 Fixed Form Statements
        - 3.3.2 Fixed Source Form
- Functions 13.7.3 Floating Point Manipulation
- Functions 13.10.12 Floating-point Manipulation
  - 8.1.1.2 Control Flow in Blocks
- Lexical Tokens, and Source Form 3 Characters,
  - 3.3 Source Form
    - 3.3.1 Free Source Form
    - 3.3.2 Fixed Source Form
      - 7.5.1.1 General Form
        - 3.3.1.1 Free Form Commentary
      - 3.3.2.1 Fixed Form Commentary
        - 10.2 Form of a Format Item List
          - 7.1.1 Form of an Expression
  - 7.1.1.7 General Form of an Expression
    - 8.1.4.1.1 Form of the Block DO Construct
    - 8.1.3.1 Form of the CASE Construct
    - 8.1.2.1 Form of the IF Construct
- Assignment 7.5.3.1 General Form of the Masked Array

# Permuted Index for Headings

8.1.4.1.2 Form of the Nonblock DO Construct  
 Statement 9.6.1.10 FORM= Specifier in the INQUIRE  
 Statement 9.3.4.4 FORM= Specifier in the OPEN  
 3.3.1.3 Free Form Statement Continuation  
 3.3.2.3 Fixed Form Statement Continuation  
 3.3.1.2 Free Form Statement Separation  
 3.3.2.2 Fixed Form Statement Separation  
 3.3.1.4 Free Form Statements  
 3.3.2.4 Fixed Form Statements  
 Between Input/Output List and Format 10.3 Interaction  
 9.4.4.3 Establishing a Format  
 10.4 Positioning by Format Control  
 10.2 Form of a Format Item List  
 10.1.2 Character Format Specification  
 10.1 Explicit Format Specification Methods  
 9.4.1.1 Format Specifier  
 10.1.1 FORMAT Statement  
 9.4.4.2 Formatted Data Transfer  
 9.1.1 Formatted Record  
 9.4.5 Printing of Formatted Records  
 INQUIRE Statement 9.6.1.11 FORMATTED= Specifier in the  
 10.8 List-Directed Formatting  
 10.9 Namelist Formatting  
 9.4.4.5 List-Directed Formatting  
 9.4.4.6 Namelist Formatting  
 8.1.4.1 Forms of the DO Construct  
 of Procedures by Means Other Than Fortran 12.5.3 Definition  
 1.4.1 FORTRAN 77 Compatibility  
 2 Fortran Terms and Concepts  
 3.3.1.1 Free Form Commentary  
 3.3.1.3 Free Form Statement Continuation  
 3.3.1.2 Free Form Statement Separation  
 3.3.1.4 Free Form Statements  
 3.3.1 Free Source Form  
 12.5.4 Statement Function  
 13.10.1 Argument Presence Inquiry Function  
 13.10.11 Transfer Function  
 13.10.17 Array Reshape Function  
 Association Status Inquiry Function 13.10.20 Pointer  
 13.10.5 Character Inquiry Function  
 13.10.7 Logical Function  
 13.10.9 Bit Inquiry Function  
 13.4 Argument Presence Inquiry Function  
 13.5.4 Character Inquiry Function  
 13.5.6 Logical Function  
 13.6 Transfer Function  
 13.8.7 Array Reshape Function  
 13.2.1 Elemental Intrinsic Function Arguments and Results  
 12.4.2 Function Reference  
 12.4.3 Elemental Intrinsic Function Reference  
 2.4.3.3 Function Reference  
 Items 9.7 Restrictions on Function References and List  
 12.2.2 Characteristics of Function Results  
 14.1.2.2 Function Results  
 12.5.2.2 Function Subprogram  
 12.1.2.4 Statement Functions  
 13.1 Intrinsic Functions  
 13.10 Generic Intrinsic Functions  
 13.10.10 Bit Manipulation Functions  
 Floating-point Manipulation Functions 13.10.12  
 Vector and Matrix Multiply Functions 13.10.13  
 13.10.14 Array Reduction Functions  
 13.10.15 Array Inquiry Functions  
 13.10.16 Array Construction Functions  
 13.10.18 Array Manipulation Functions  
 13.10.19 Array Location Functions  
 13.10.2 Numeric Functions  
 13.10.3 Mathematical Functions  
 13.10.4 Character Functions  
 13.10.6 Kind Functions  
 13.10.8 Numeric Inquiry Functions  
 Specific Names for Intrinsic Functions 13.12  
 13.5.1 Numeric Functions  
 13.5.2 Mathematical Functions  
 13.5.3 Character Functions  
 13.5.5 Kind Functions  
 Numeric Manipulation and Inquiry Functions 13.7  
 13.7.2 Numeric Inquiry Functions  
 Floating Point Manipulation Functions 13.7.3  
 13.8 Array Intrinsic Functions  
 Association Status Inquiry Functions 13.8.10 Pointer

|                                   |                                  |
|-----------------------------------|----------------------------------|
| Vector and Matrix Multiplication  | Functions 13.8.3                 |
| 13.8.4 Array Reduction            | Functions                        |
| 13.8.5 Array Inquiry              | Functions                        |
| 13.8.6 Array Construction         | Functions                        |
| 13.8.8 Array Manipulation         | Functions                        |
| 13.8.9 Array Location             | Functions                        |
| 2.5                               | Fundamental Terms                |
| 7.5.1.1                           | General Form                     |
| 7.1.1.7                           | General Form of an Expression    |
| Assignment 7.5.3.1                | General Form of the Masked Array |
| 10.5.4.3                          | Generalized Character Editing    |
| 10.5.4                            | Generalized Editing              |
| 10.5.4.1.1                        | Generalized Integer Editing      |
| 10.5.4.2                          | Generalized Logical Editing      |
| 10.5.4.1                          | Generalized Numeric Editing      |
| Editing 10.5.4.1.2                | Generalized Real and Complex     |
| to Names Established to be        | Generic /Procedure References    |
| 13.10                             | Generic Intrinsic Functions      |
| 14.1.2.3 Unambiguous              | Generic Procedure References     |
| 11.3.3.4                          | Global Allocatable Arrays        |
| 11.3.3.2                          | Global Data                      |
| 14.1.1                            | Global Entities                  |
| 8.1.1 Rules                       | Governing Blocks                 |
| 10.9.1.3 Namelist                 | Group Object List Items          |
| 10.9.1.1 Namelist                 | Group Object Names               |
| 10.7.2                            | H Editing                        |
| 2.1                               | High Level Syntax                |
| 12.1.2.2.1                        | Host Association                 |
| 14.6.1.2 Use Association and      | Host Association                 |
| Association 12.1.2.2.2            | Host Association and USE         |
| 11.3.3.1                          | Identical Common Blocks          |
| 9.4.4.2                           | Identifying a Unit               |
| 12.3.1                            | Implicit and Explicit Interfaces |
| 12.3.2.4                          | Implicit Interface Specification |
| 5.3                               | IMPLICIT Statement               |
| 8.1.4.3 Active and                | Inactive DO Constructs           |
| 3.4                               | Including Source Text            |
| 1.3.1                             | Inclusions                       |
| 1.3                               | Inclusions and Exclusions        |
| Associated with Alternate Return  | Indicators 12.4.1.3 Arguments    |
| 9.4.1 Control                     | Information List                 |
| 14.7.3 Variables That Are         | Initially Defined                |
| 14.7.4 Variables That Are         | Initially Undefined              |
| 8.1.4.4.1 Loop                    | Initiation                       |
| 10.8.1 List-Directed              | Input                            |
| 10.9.1 Namelist                   | Input                            |
| 10.8.1.2 List-Directed            | Input Example                    |
| 10.9.1.6 Namelist                 | Input Example                    |
| 10.9.1.2 Namelist                 | Input Values                     |
| Advancing and Nonadvancing        | Input/Output 9.2.1.3.1           |
| 10                                | Input/Output Editing             |
| 9.4.2 Data Transfer               | Input/Output List                |
| 10.3 Interaction Between          | Input/Output List and Format     |
| Execution of a Data Transfer      | Input/Output Statement 9.4.4     |
| 9                                 | Input/Output Statements          |
| 9.8 Restriction on                | Input/Output Statements          |
| 9.4.1.4                           | Input/Output Status              |
| 14.3 Scope of External            | Input/Output Units               |
| 9.6.3                             | Inquire by Output List           |
| 9.6.1.1 FILE= Specifier in the    | INQUIRE Statement                |
| 9.6.1.10 FORM= Specifier in the   | INQUIRE Statement                |
| FORMATTED= Specifier in the       | INQUIRE Statement 9.6.1.11       |
| UNFORMATTED= Specifier in the     | INQUIRE Statement 9.6.1.12       |
| 9.6.1.13 RECL= Specifier in the   | INQUIRE Statement                |
| NEXTREC= Specifier in the         | INQUIRE Statement 9.6.1.14       |
| 9.6.1.15 BLANK= Specifier in the  | INQUIRE Statement                |
| POSITION= Specifier in the        | INQUIRE Statement 9.6.1.16       |
| 9.6.1.17 ACTION= Specifier in the | INQUIRE Statement                |
| 9.6.1.18 READ= Specifier in the   | INQUIRE Statement                |
| 9.6.1.19 WRITE= Specifier in the  | INQUIRE Statement                |
| 9.6.1.2 EXIST= Specifier in the   | INQUIRE Statement                |
| READWRITE= Specifier in the       | INQUIRE Statement 9.6.1.20       |
| 9.6.1.21 DELIM= Specifier in the  | INQUIRE Statement                |
| 9.6.1.22 PAD= Specifier in the    | INQUIRE Statement                |
| 9.6.1.3 OPENED= Specifier in the  | INQUIRE Statement                |
| 9.6.1.4 NUMBER= Specifier in the  | INQUIRE Statement                |
| 9.6.1.5 NAMED= Specifier in the   | INQUIRE Statement                |
| 9.6.1.6 NAME= Specifier in the    | INQUIRE Statement                |
| 9.6.1.7 ACCESS= Specifier in the  | INQUIRE Statement                |
| SEQUENTIAL= Specifier in the      | INQUIRE Statement 9.6.1.8        |
| 9.6.1.9 DIRECT= Specifier in the  | INQUIRE Statement                |

# Permuted Index for Headings

9.6 File Inquiry  
 13.10.1 Argument Presence Inquiry Function  
 Pointer Association Status Inquiry Function 13.10.20  
 13.10.5 Character Inquiry Function  
 13.10.9 Bit Inquiry Function  
 13.4 Argument Presence Inquiry Function  
 13.5.4 Character Inquiry Function  
 13.10.15 Array Inquiry Functions  
 13.10.8 Numeric Inquiry Functions  
 13.7 Numeric Manipulation and Inquiry Functions  
 13.7.2 Numeric Inquiry Functions  
 Pointer Association Status Inquiry Functions 13.8.10  
 13.8.5 Array Inquiry Functions  
 13.5.7 Bit Manipulation and Inquiry Procedures  
 9.6.1 Inquiry Specifiers  
 9.6.2 Restrictions on Inquiry Specifiers  
 12.5.2.4 Instances of a Subprogram  
 5.1.1.1 INTEGER  
 13.7.1 Models for Integer and Real Data  
 7.2.1.1 Integer Division  
 10.5.1.1 Integer Editing  
 10.5.4.1.1 Generalized Integer Editing  
 4.3.1.1 Integer Type  
 7.1.7.2 Integrity of Parentheses  
 5.1.2.3 INTENT Attribute  
 12.5.2.1 Effects of INTENT Attribute on Subprograms  
 5.2.1 INTENT Statement  
 List and Format 10.3 Interaction Between Input/Output  
 12.3 Procedure Interface  
 12.3.1.1 Explicit Interface  
 Specification of the Procedure Interface 12.3.2  
 12.3.2.1 Procedure Interface Block  
 2.2.3.4 Procedure Interface Block  
 12.3.2.4 Implicit Interface Specification  
 12.3.1 Implicit and Explicit Interfaces  
 12.1.2.2 External, Internal, and Module Procedures  
 9.2.2.1 Internal File Properties  
 9.2.2.2 Internal File Restrictions  
 9.2.2 Internal Files  
 2.2.3.3 Internal Procedure  
 11.1.3 Main Program Internal Procedures  
 Assignment Statements 7.5.1.6 Interpretation of Defined  
 Operations 7.3 Interpretation of Defined  
 Assignments 7.5.1.5 Interpretation of Intrinsic  
 Operations 7.2 Interpretation of Intrinsic  
 Assignments 7.5.3.2 Interpretation of Masked Array  
 2.5.7 Intrinsic  
 4 Intrinsic and Derived Data Types  
 Rules 7.5.1.4 Intrinsic Assignment Conformance  
 7.5.1.2 Intrinsic Assignment Statement  
 7.5.1.5 Interpretation of Intrinsic Assignments  
 5.1.2.1.1 INTRINSIC Attribute  
 4.3 Intrinsic Data Types  
 Results 13.2.1 Elemental Intrinsic Function Arguments and  
 12.4.3 Elemental Intrinsic Function Reference  
 13.1 Intrinsic Functions  
 13.10 Generic Intrinsic Functions  
 13.12 Specific Names for Intrinsic Functions  
 13.8 Array Intrinsic Functions  
 Evaluation of the Character Intrinsic Operation 7.1.7.4  
 7.2.2 Character Intrinsic Operation  
 7.1.2 Intrinsic Operations  
 7.1.5 Conformability Rules for Intrinsic Operations  
 7.1.7.3 Evaluation of Numeric Intrinsic Operations  
 7.1.7.5 Evaluation of Relational Intrinsic Operations  
 7.1.7.6 Evaluation of Logical Intrinsic Operations  
 7.2 Interpretation of Intrinsic Operations  
 7.2.1 Numeric Intrinsic Operations  
 7.2.3 Relational Intrinsic Operations  
 7.2.4 Logical Intrinsic Operations  
 12.5.1 Intrinsic Procedure Definition  
 12.1.2.1 Intrinsic Procedures  
 13 Intrinsic Procedures  
 13.13 Specifications of the Intrinsic Procedures  
 13.2 Elemental Intrinsic Procedures  
 12.3.2.3 INTRINSIC Statement  
 13.2.2 Elemental Intrinsic Subroutine Arguments  
 12.4.5 Elemental Intrinsic Subroutine Reference  
 13.11 Intrinsic Subroutines  
 13.9 Intrinsic Subroutines  
 2.4.1.1 Intrinsic Type

10.2 Form of a Format Item List  
 Namelist Group Object List Items 10.9.1.3  
 on Function References and List Items 9.7 Restrictions  
     2.5.2 Keyword  
 Positional Arguments or Argument Keywords 13.3  
     14.1.2.6 Argument Keywords  
         3.2.1 Keywords  
         13.10.6 Kind Functions  
         13.5.5 Kind Functions  
 /Numeric, Mathematical, Character, Kind, Logical, and Bit Procedures  
     14.2 Scope of Labels  
     3.2.5 Statement Labels  
     8.2.1 Statement Labels  
         3.1.1 Letters  
     2.1 High Level Syntax  
         7.1.1.2 Level-1 Expressions  
         7.1.1.3 Level-2 Expressions  
         7.1.1.4 Level-3 Expressions  
         7.1.1.5 Level-4 Expressions  
         7.1.1.6 Level-5 Expressions  
     3 Characters, Lexical Tokens, and Source Form  
     11.3.3.5 Procedure Libraries  
 10.2 Form of a Format Item List  
     12.4.1 Actual Argument List  
     9.4.1 Control Information List  
 9.4.2 Data Transfer Input/Output List  
     9.6.3 Inquire by Output List  
 Interaction Between Input/Output List and Format 10.3  
     10.9.1.3 Namelist Group Object List Items  
     on Function References and List Items 9.7 Restrictions  
         10.8 List-Directed Formatting  
         9.4.4.5 List-Directed Formatting  
         10.8.1 List-Directed Input  
         10.8.1.2 List-Directed Input Example  
         10.8.2 List-Directed Output  
         14.1.2 Local Entities  
         13.10.19 Array Location Functions  
         13.8.9 Array Location Functions  
         5.1.1.6 LOGICAL  
 /Mathematical, Character, Kind, Logical, and Bit Procedures  
     10.5.2 Logical Editing  
     10.5.4.2 Generalized Logical Editing  
         13.10.7 Logical Function  
         13.5.6 Logical Function  
     7.1.7.6 Evaluation of Logical Intrinsic Operations  
         7.2.4 Logical Intrinsic Operations  
         4.3.2.2 Logical Type  
         8.1.4.4.1 Loop Initiation  
         8.1.4.4.4 Loop Termination  
         3.2 Low-Level Syntax  
         11.1 Main Program  
         2.2.2 Main Program  
         11.1.2 Main Program Executable Part  
         11.1.3 Main Program Internal Procedures  
         11.1.1 Main Program Specifications  
     Functions 13.7 Numeric Manipulation and Inquiry  
     Procedures 13.5.7 Bit Manipulation and Inquiry  
         13.10.10 Bit Manipulation Functions  
     13.10.12 Floating-point Manipulation Functions  
         13.10.18 Array Manipulation Functions  
     13.7.3 Floating Point Manipulation Functions  
         13.8.8 Array Manipulation Functions  
         13.8.2 Mask Arguments  
     7.5.3.1 General Form of the Masked Array Assignment  
         7.5.3 Masked Array Assignment WHERE  
     7.5.3.2 Interpretation of Masked Array Assignments  
 Logical, and Bit/ 13.5 Numeric, Mathematical, Character, Kind,  
     13.10.3 Mathematical Functions  
     13.5.2 Mathematical Functions  
     13.8.3 Vector and Matrix Multiplication Functions  
     13.10.13 Vector and Matrix Multiply Functions  
     Procedure Classification by Means of Definition 12.1.2  
     Definition of Procedures by Means Other Than Fortran 12.5.3  
     Explicit Format Specification Methods 10.1  
         13.7.1 Models for Integer and Real Data  
         2.2.4 Module  
         2.2.3.2 Module Procedure  
 12.1.2.2 External, Internal, and Module Procedures  
     11.3.1 Module Reference  
     11.3 Modules  
 11.3.3 Examples of the Use of Modules

|                                    |                             |                                         |
|------------------------------------|-----------------------------|-----------------------------------------|
|                                    | 1.7                         | Modules                                 |
| 13.8.3                             | Vector and Matrix           | Multiplication Functions                |
| 13.10.13                           | Vector and Matrix           | Multiply Functions                      |
|                                    | 2.5.1                       | Name and Designator                     |
|                                    | 14.6.1                      | Name Association                        |
| Statement                          | 9.6.1.6                     | NAME= Specifier in the INQUIRE          |
| 5.5.2.4                            | Differences between         | Named Common and Blank Common           |
| Statement                          | 9.6.1.5                     | NAMED= Specifier in the INQUIRE         |
|                                    | 10.9                        | Namelist Formatting                     |
|                                    | 9.4.4.6                     | Namelist Formatting                     |
|                                    | 10.9.1.3                    | Namelist Group Object List Items        |
|                                    | 10.9.1.1                    | Namelist Group Object Names             |
|                                    | 10.9.1                      | Namelist Input                          |
|                                    | 10.9.1.6                    | Namelist Input Example                  |
|                                    | 10.9.1.2                    | Namelist Input Values                   |
|                                    | 10.9.2                      | Namelist Output                         |
|                                    | 10.9.2.1                    | Namelist Output Editing                 |
|                                    | 10.9.2.2                    | Namelist Output Records                 |
|                                    | 9.4.1.2                     | Namelist Specifier                      |
|                                    | 5.4                         | NAMELIST Statement                      |
| 10.9.1.1                           | Namelist Group Object       | Names                                   |
|                                    | 14.1                        | Scope of Names                          |
|                                    | 3.2.2                       | Names                                   |
| Designators                        | 5.5.1.3                     | Array Names and Array Element           |
| /Resolving Procedure References to |                             | Names Established to be Generic         |
| /Resolving Procedure References to |                             | Names Established to be Only/           |
|                                    | 13.12                       | Specific Names for Intrinsic Functions  |
| Resolving Procedure References to  |                             | Names Not Established 14.1.2.4.3        |
|                                    | 1.6.1                       | Nature of Deleted Features              |
|                                    | 1.6.2                       | Nature of Obsolescent Features          |
| Statement                          | 9.6.1.14                    | NEXTREC= Specifier in the INQUIRE       |
| 9.2.1.3.1                          | Advancing and               | Nonadvancing Input/Output               |
| 8.1.4.1.2                          | Form of the                 | Nonblock DO Construct                   |
|                                    | 3.3.1.3.1                   | Noncharacter Context Continuation       |
|                                    | 4.3.2                       | Nonnumeric Types                        |
|                                    | 1.8                         | Normative References                    |
|                                    | 1.5                         | Notation Used in This Standard          |
|                                    | 10.8.1.1                    | Null Values                             |
|                                    | 10.9.1.4                    | Null Values                             |
|                                    | 6.3.2                       | NULLIFY Statement                       |
|                                    | 9.4.1.3                     | Record Number                           |
| Statement                          | 9.6.1.4                     | NUMBER= Specifier in the INQUIRE        |
| 13.9.2                             | Pseudorandom                | Numbers                                 |
|                                    | 10.5.1                      | Numeric Editing                         |
| 10.5.4.7                           | Generalized                 | Numeric Editing                         |
|                                    | 13.10.2                     | Numeric Functions                       |
|                                    | 13.5.1                      | Numeric Functions                       |
|                                    | 13.10.8                     | Numeric Inquiry Functions               |
|                                    | 13.7.2                      | Numeric Inquiry Functions               |
| 7.1.7.3                            | Evaluation of               | Numeric Intrinsic Operations            |
|                                    | 7.2.1                       | Numeric Intrinsic Operations            |
| Functions                          | 13.7                        | Numeric Manipulation and Inquiry        |
| Kind, Logical, and Bit/            | 13.5                        | Numeric, Mathematical, Character,       |
|                                    | 4.3.1                       | Numeric Types                           |
|                                    | 2.4.3.1                     | Data Object                             |
| Specifications                     | 5                           | Data Object Declarations and            |
| 10.9.1.3                           | Namelist Group              | Object List Items                       |
| 10.9.1.1                           | Namelist Group              | Object Names                            |
| Characteristics of Dummy Data      |                             | Objects 12.2.1.1                        |
| Associated with Dummy Data         |                             | Objects 12.4.1.1 Arguments              |
| Association of Scalar Data         |                             | Objects 14.6.3.3                        |
| of Types and Values to             |                             | Objects 4.2 Relationship                |
| 5.5                                | Storage Association of Data | Objects                                 |
| 5.5.1.2                            | Equivalence of Character    | Objects                                 |
|                                    | 6                           | Use of Data Objects                     |
|                                    | 14.7.1                      | Definition of Objects and Subobjects    |
|                                    | 1.6                         | Deleted and Obsolescent Features        |
|                                    | 1.6.2                       | Nature of Obsolescent Features          |
|                                    | 9.3.4                       | The OPEN Statement                      |
|                                    | 9.3.4.1                     | FILE= Specifier in the OPEN Statement   |
|                                    | 9.3.4.10                    | PAD= Specifier in the OPEN Statement    |
|                                    | 9.3.4.2                     | STATUS= Specifier in the OPEN Statement |
|                                    | 9.3.4.3                     | ACCESS= Specifier in the OPEN Statement |
|                                    | 9.3.4.4                     | FORM= Specifier in the OPEN Statement   |
|                                    | 9.3.4.5                     | RECL= Specifier in the OPEN Statement   |
|                                    | 9.3.4.6                     | BLANK= Specifier in the OPEN Statement  |
|                                    | POSITION=                   | Specifier in the OPEN Statement 9.3.4.7 |
| 9.3.4.8                            | ACTION=                     | Specifier in the OPEN Statement         |
| 9.3.4.9                            | DELIM=                      | Specifier in the OPEN Statement         |
| Statement                          | 9.6.1.3                     | OPENED= Specifier in the INQUIRE        |
| 7.1.7.1                            | Evaluation of               | Operands                                |



and Shape of the Result of an Operation /Type, Type Parameters,  
of the Character Intrinsic Operation 7.1.7.4 Evaluation  
7.1.7.7 Evaluation of a Defined Operation  
7.2.2 Character Intrinsic Operation  
7.3.1 Unary Defined Operation  
7.3.2 Binary Defined Operation  
12.3.2.1.1 Defined Operations  
4.1.3 Operations  
7.1.2 Intrinsic Operations  
7.1.3 Defined Operations  
Rules for Intrinsic Operations 7.1.5 Conformability  
7.1.7 Evaluation of Operations  
Evaluation of Numeric Intrinsic Operations 7.1.7.3  
of Relational Intrinsic Operations 7.1.7.5 Evaluation  
Evaluation of Logical Intrinsic Operations 7.1.7.6  
7.2 Interpretation of Intrinsic Operations  
7.2.1 Numeric Intrinsic Operations  
7.2.3 Relational Intrinsic Operations  
7.2.4 Logical Intrinsic Operations  
7.3 Interpretation of Defined Operations  
4.4.5 Derived-Type Operations and Assignment  
2.5.8 Operator  
11.3.3.6 Operator Extensions  
14.4 Scope of Operators  
3.2.4 Operators  
7.4 Precedence of Operators  
5.1.2.6 OPTIONAL Attribute  
5.2.2 OPTIONAL Statement  
2.3.2 Statement Order  
6.2.2.2 Array Element Order  
10.8.2 List-Directed Output  
10.9.2 Namelist Output  
10.9.2.1 Namelist Output Editing  
9.6.3 Inquire by Output List  
10.9.2.2 Namelist Output Records  
10.6.5 P Editing  
Statement 9.6.1.22 PAD= Specifier in the INQUIRE  
Statement 9.3.4.10 PAD= Specifier in the OPEN  
5.1.2.1 PARAMETER Attribute  
5.2.10 PARAMETER Statement  
Primary 7.1.4.1 Data Type, Type Parameters, and Shape of a  
Expression 7.1.4 Data Type, Type Parameters, and Shape of an  
Result/ 7.1.4.2 Data Type, Type Parameters, and Shape of the  
7.1.7.2 Integrity of Parentheses  
11.1.2 Main Program Executable Part  
8.5 PAUSE Statement  
13.7.3 Floating Point Manipulation Functions  
2.4.6 Pointer  
7.5.2 Pointer Assignment  
14.6.2 Pointer Association  
14.6.2.1 Pointer Association Status  
Inquiry Function 13.10.20 Pointer Association Status  
Inquiry Functions 13.8.10 Pointer Association Status  
5.1.2.7 POINTER Attribute  
14.6.2.2 Pointer Definition Status  
5.2.7 POINTER Statement  
6.3.1.2 Allocation of Pointer Targets  
6.3.3.2 Deallocation of Pointer Targets  
9.2.1.3 File Position  
9.2.1.3.3 File Position after Data Transfer  
10.6.1 Position Editing  
9.2.1.3.2 File Position Prior to Data Transfer  
INQUIRE Statement 9.6.1.16 POSITION= Specifier in the  
Statement 9.3.4.7 POSITION= Specifier in the OPEN  
Keywords 13.3 Positional Arguments or Argument  
10.4 Positioning by Format Control  
9.5 File Positioning Statements  
7.4 Precedence of Operators  
5.1.1.3 DOUBLE PRECISION  
9.3.3 Preconnection  
13.10.1 Argument Presence Inquiry Function  
13.4 Argument Presence Inquiry Function  
on Dummy Arguments Not Present 12.5.2.8 Restrictions  
7.1.1.1 Primary  
Type Parameters, and Shape of a Primary 7.1.4.1 Data Type,  
9.4.5 Printing of Formatted Records  
9.2.1.3.2 File Position Prior to Data Transfer  
2.2.3 Procedure  
2.2.3.1 External Procedure  
2.2.3.2 Module Procedure  
2.2.3.3 Internal Procedure

# Permuted Index for Headings

|                                   |            |                                   |
|-----------------------------------|------------|-----------------------------------|
| of Definition                     | 12.1.2     | Procedure Classification by Means |
| Reference                         | 12.1.1     | Procedure Classification by       |
|                                   | 12.1       | Procedure Classifications         |
|                                   | 12.5       | Procedure Definition              |
| 12.5.1 Intrinsic                  |            | Procedure Definition              |
|                                   | 12.3       | Procedure Interface               |
| 12.3.2 Specification of the       |            | Procedure Interface               |
|                                   | 12.3.2.1   | Procedure Interface Block         |
|                                   | 2.2.3.4    | Procedure Interface Block         |
|                                   | 11.3.3.5   | Procedure Libraries               |
|                                   | 12.4       | Procedure Reference               |
| 14.1.2.3 Unambiguous Generic      |            | Procedure References              |
|                                   | 14.1.2.4   | Procedure References              |
| Established/                      | 14.1.2.4.1 | Procedure References to Names     |
| Established/                      | 14.1.2.4.2 | Procedure References to Names     |
| Established                       | 14.1.2.4.3 | Procedure References to Names Not |
| 11.1.3 Main Program Internal      |            | Procedures                        |
|                                   | 12         | Procedures                        |
|                                   | 12.1.2.1   | Procedures                        |
| External, Internal, and Module    |            | Procedures 12.1.2.2               |
|                                   | 12.1.2.3   | Procedures                        |
| 12.2 Characteristics of           |            | Procedures                        |
| 12.2.1.2 Characteristics of Dummy |            | Procedures                        |
| Arguments Associated with Dummy   |            | Procedures 12.4.1.2               |
|                                   | 13         | Procedures                        |
| Specifications of the Intrinsic   |            | Procedures 13.13                  |
| 13.2 Elemental Intrinsic          |            | Procedures                        |
| Character, Kind, Logical, and Bit |            | Procedures /Mathematical,         |
| Bit Manipulation and Inquiry      |            | Procedures 13.5.7                 |
| Fortran                           | 12.5.3     | Definition of                     |
|                                   | 12.5.2     | Procedures Defined by Subprograms |
|                                   | 1.2        | Processor                         |
|                                   | 3.1        | Processor Character Set           |
| 9.2.2.1 Internal File             |            | Properties                        |
|                                   | 13.9.2     | Pseudorandom Numbers              |
|                                   | 11.3.3.8   | Public Entities Renamed           |
|                                   | 1.1        | Purpose                           |
|                                   | 8.1.4.2    | Range of the DO Construct         |
| Statement                         | 9.6.1.18   | READ= Specifier in the INQUIRE    |
| INQUIRE Statement                 | 9.6.1.20   | READWRITE= Specifier in the       |
|                                   | 5.1.1.2    | REAL                              |
|                                   | 10.5.1.2   | Real and Complex Editing          |
| 10.5.4.1.2 Generalized            |            | Real and Complex Editing          |
| 13.7.1 Models for Integer and     |            | Real Data                         |
|                                   | 4.3.1.2    | Real Type                         |
| Statement                         | 9.6.1.13   | RECL= Specifier in the INQUIRE    |
| Statement                         | 9.3.4.5    | RECL= Specifier in the OPEN       |
| 9.1.1 Formatted                   |            | Record                            |
| 9.1.2 Unformatted                 |            | Record                            |
| 9.1.3 Endfile                     |            | Record                            |
| 9.4.1.3                           |            | Record Number                     |
| 10.9.2.2 Namelist Output          |            | Records                           |
|                                   | 9.1        | Records                           |
| 9.4.5 Printing of Formatted       |            | Records                           |
|                                   | 13.10.14   | Array Reduction Functions         |
|                                   | 13.8.4     | Array Reduction Functions         |
|                                   | 11.3.1     | Module Reference                  |
| Procedure Classification by       |            | Reference 12.1.1                  |
|                                   | 12.4       | Procedure Reference               |
|                                   | 12.4.2     | Function Reference                |
| Elemental Intrinsic Function      |            | Reference 12.4.3                  |
|                                   | 12.4.4     | Subroutine Reference              |
| Elemental Intrinsic Subroutine    |            | Reference 12.4.5                  |
|                                   | 2.4.3.3    | Function Reference                |
|                                   | 2.5.5      | Reference                         |
| Unambiguous Generic Procedure     |            | References 14.1.2.3               |
| 14.1.2.4 Resolving Procedure      |            | References                        |
|                                   | 1.8        | Normative References              |
| 9.7 Restrictions on Function      |            | References and List Items         |
| 14.1.2.4.1 Resolving Procedure    |            | References to Names Established/  |
| 14.1.2.4.2 Resolving Procedure    |            | References to Names Established/  |
| 14.1.2.4.3 Resolving Procedure    |            | References to Names Not/          |
| 7.1.7.5 Evaluation of             |            | Relational Intrinsic Operations   |
|                                   | 7.2.3      | Relational Intrinsic Operations   |
| Status and Definition/            | 14.6.2.3   | Relationship Between Association  |
| to Objects                        | 4.2        | Relationship of Types and Values  |
| 11.3.3.8 Public Entities          |            | Renamed                           |
|                                   | 13.10.17   | Array Reshape Function            |
|                                   | 13.8.7     | Array Reshape Function            |
|                                   | 14.1.2.4   | Resolving Procedure References    |
| Names Established to/             | 14.1.2.4.1 | Resolving Procedure References to |

|                                   |                           |                                             |
|-----------------------------------|---------------------------|---------------------------------------------|
| Names Established to/             | 14.1.2.4.2                | Resolving Procedure References to           |
| Names Not Established             | 14.1.2.4.3                | Resolving Procedure References to           |
|                                   | Statements                | 9.8                                         |
|                                   | 9.2.2.2                   | Internal File Restrictions                  |
|                                   | Equivalence               | 5.5.2.5                                     |
|                                   | Not Present               | 12.5.2.8                                    |
| Associated with Dummy/            | 12.5.2.9                  | Restrictions on Dummy Arguments             |
|                                   | Statements                | 5.5.1.4                                     |
|                                   | References and List Items | 9.7                                         |
|                                   | Specifiers                | 9.6.2                                       |
| Type Parameters, and Shape of the |                           | Characteristics of Function                 |
|                                   | 14.1.2.2                  | Function Results                            |
|                                   | 14.1.2.2                  | Function Results                            |
|                                   | 13.2.1                    | Elemental Results                           |
| Associated with Alternate         |                           | Return Indicators /Arguments                |
|                                   | 12.5.2.6                  | RETURN Statement                            |
|                                   | 9.5.3                     | REWIND Statement                            |
|                                   | 1.5.1                     | Syntax Rules                                |
| 1.5.2                             |                           | Assumed Syntax Rules                        |
| Intrinsic Assignment Conformance  |                           | Rules                                       |
|                                   | 7.1.5                     | Conformability Rules                        |
|                                   | 8.1.1                     | Rules for Intrinsic Operations              |
|                                   | 8.1.1                     | Rules Governing Blocks                      |
|                                   | 10.6.4                    | S, SP, and SS Editing                       |
|                                   | 5.1.2.5                   | SAVE Attribute                              |
|                                   | 5.2.4                     | SAVE Statement                              |
|                                   | 2.4.4                     | Scalar                                      |
|                                   | 7.1.6                     | Scalar and Array Expressions                |
| 14.6.3.3                          |                           | Association of Scalar Data Objects          |
|                                   | 6.1                       | Scalars                                     |
|                                   | 10.6.5.1                  | Scale Factor                                |
| Definition                        | 14                        | Scope, Association, and                     |
| Units                             | 14.3                      | Scope of External Input/Output              |
|                                   | 14.2                      | Scope of Labels                             |
|                                   | 14.1                      | Scope of Names                              |
|                                   | 14.4                      | Scope of Operators                          |
|                                   | 1                         | Scope of Standard                           |
|                                   | 14.5                      | Scope of the Assignment Symbol              |
| 6.2.2                             |                           | Array Elements and Array Sections           |
|                                   | 6.2.2.3                   | Array Sections                              |
|                                   | 3.3.1.2                   | Free Form Statement Separation              |
|                                   | 3.3.2.2                   | Fixed Form Statement Separation             |
|                                   | 14.6.3.1                  | Storage Sequence                            |
|                                   | 2.3.4                     | Execution Sequence                          |
|                                   | 2.5.9                     | Sequence                                    |
|                                   | 4.3.2.1.1                 | Collating Sequence                          |
| 5.5.2.1                           |                           | Common Block Storage Sequence               |
|                                   | 12.4.1.4                  | Sequence Association                        |
| 14.6.3.2                          |                           | Association of Storage Sequences            |
|                                   | 9.2.1.2.1                 | Sequential Access                           |
| INQUIRE Statement                 | 9.6.1.8                   | SEQUENTIAL= Specifier in the                |
| 3.1                               |                           | Processor Character Set                     |
|                                   | 4.1.1                     | Set of Values                               |
| Data Type, Type Parameters, and   |                           | Shape of a Primary                          |
| Data Type, Type Parameters, and   |                           | Shape of an Expression                      |
|                                   | 13.8.1                    | The Shape of Array Arguments                |
| /Data Type, Type Parameters, and  |                           | Shape of the Result of an/                  |
|                                   | 5.5.2.2                   | Size of a Common Block                      |
|                                   | 10.6.2                    | Slash Editing                               |
| 3                                 |                           | Characters, Lexical Tokens, and             |
|                                   | 3.3                       | Source Form                                 |
|                                   | 3.3.1                     | Free Source Form                            |
|                                   | 3.3.2                     | Fixed Source Form                           |
|                                   | 3.4                       | Including Source Text                       |
|                                   | 10.6.4                    | S, SP, and SS Editing                       |
|                                   | 3.1.4                     | Special Characters                          |
| to Names Established to be Only   |                           | Specific /Procedure References              |
|                                   | 10.1.2                    | Character Format Specification              |
|                                   | 12.3.2.4                  | Implicit Interface Specification            |
|                                   | 7.1.6.2                   | Specification Expression                    |
|                                   | 10.1                      | Explicit Format Specification Methods       |
|                                   | Interface                 | 12.3.2                                      |
|                                   | 5.2                       | Attribute Specification Statements          |
|                                   | 11.1.1                    | Main Program Specifications                 |
| 5                                 |                           | Data Object Declarations and Specifications |
|                                   | 13.13                     | Procedures Specifications of the Intrinsic  |
|                                   | 9.4.1.1                   | Format Specifier                            |
|                                   | 9.4.1.2                   | Namelist Specifier                          |
|                                   | 9.4.1.8                   | Advance Specifier                           |
|                                   | 9.3.5.1                   | STATUS= Specifier in the CLOSE Statement    |
| Statement                         | 9.6.1.1                   | FILE= Specifier in the INQUIRE              |

# Permuted Index for Headings

|                                   |                                 |
|-----------------------------------|---------------------------------|
| Statement 9.6.1.10 FORM=          | Specifier in the INQUIRE        |
| Statement 9.6.1.11 FORMATTED=     | Specifier in the INQUIRE        |
| Statement 9.6.1.12 UNFORMATTED=   | Specifier in the INQUIRE        |
| Statement 9.6.1.13 RECL=          | Specifier in the INQUIRE        |
| Statement 9.6.1.14 NEXTREC=       | Specifier in the INQUIRE        |
| Statement 9.6.1.15 BLANK=         | Specifier in the INQUIRE        |
| Statement 9.6.1.16 POSITION=      | Specifier in the INQUIRE        |
| Statement 9.6.1.17 ACTION=        | Specifier in the INQUIRE        |
| Statement 9.6.1.18 READ=          | Specifier in the INQUIRE        |
| Statement 9.6.1.19 WRITE=         | Specifier in the INQUIRE        |
| Statement 9.6.1.2 EXIST=          | Specifier in the INQUIRE        |
| Statement 9.6.1.20 READWRITE=     | Specifier in the INQUIRE        |
| Statement 9.6.1.21 DELIM=         | Specifier in the INQUIRE        |
| Statement 9.6.1.22 PAD=           | Specifier in the INQUIRE        |
| Statement 9.6.1.3 OPENED=         | Specifier in the INQUIRE        |
| Statement 9.6.1.4 NUMBER=         | Specifier in the INQUIRE        |
| Statement 9.6.1.5 NAMED=          | Specifier in the INQUIRE        |
| Statement 9.6.1.6 NAME=           | Specifier in the INQUIRE        |
| Statement 9.6.1.7 ACCESS=         | Specifier in the INQUIRE        |
| Statement 9.6.1.8 SEQUENTIAL=     | Specifier in the INQUIRE        |
| Statement 9.6.1.9 DIRECT=         | Specifier in the INQUIRE        |
| 9.3.4.1 FILE=                     | Specifier in the OPEN Statement |
| 9.3.4.10 PAD=                     | Specifier in the OPEN Statement |
| 9.3.4.2 STATUS=                   | Specifier in the OPEN Statement |
| 9.3.4.3 ACCESS=                   | Specifier in the OPEN Statement |
| 9.3.4.4 FORM=                     | Specifier in the OPEN Statement |
| 9.3.4.5 RECL=                     | Specifier in the OPEN Statement |
| 9.3.4.6 BLANK=                    | Specifier in the OPEN Statement |
| 9.3.4.7 POSITION=                 | Specifier in the OPEN Statement |
| 9.3.4.8 ACTION=                   | Specifier in the OPEN Statement |
| 9.3.4.9 DELIM=                    | Specifier in the OPEN Statement |
| 5.1.1 Type                        | Specifiers                      |
| 9.6.1 Inquiry                     | Specifiers                      |
| 9.6.2 Restrictions on Inquiry     | Specifiers                      |
| 10.6.4 S, SP, and                 | SS Editing                      |
| 1 Scope of                        | Standard                        |
| 1.5 Notation Used in This         | Standard                        |
| 2.3.1 Executable/Nonexecutable    | Statements                      |
| 3.3.1.4 Free Form                 | Statements                      |
| 3.3.2.4 Fixed Form                | Statements                      |
| 5.1 Type Declaration              | Statements                      |
| 5.2 Attribute Specification       | Statements                      |
| 5.2.3 Accessibility               | Statements                      |
| Restrictions on EQUIVALENCE       | Statements 5.5.1.4              |
| of Defined Assignment             | Statements /Interpretation      |
| 9 Input/Output                    | Statements                      |
| 9.4 Data Transfer                 | Statements                      |
| Termination of Data Transfer      | Statements 9.4.6                |
| 9.5 File Positioning              | Statements                      |
| 9.8 Restriction on Input/Output   | Statements                      |
| 14.6.2.1 Pointer Association      | Status                          |
| 14.6.2.2 Pointer Definition       | Status                          |
| Association Status and Definition | Status /Relationship Between    |
| 14.8 Allocation                   | Status                          |
| 9.4.1.4 Input/Output              | Status                          |
| /Relationship Between Association | Status and Definition Status    |
| 13.10.20 Pointer Association      | Status Inquiry Function         |
| 13.8.10 Pointer Association       | Status Inquiry Functions        |
| Statement 9.3.5.1                 | STATUS= Specifier in the CLOSE  |
| Statement 9.3.4.2                 | STATUS= Specifier in the OPEN   |
| 8.4                               | STOP Statement                  |
| 2.4.7                             | Storage                         |
| 14.6.3                            | Storage Association             |
| Objects 5.5                       | Storage Association of Data     |
| 14.6.3.1                          | Storage Sequence                |
| 5.5.2.1 Common Block              | Storage Sequence                |
| 14.6.3.2 Association of           | Storage Sequences               |
| 10.7 Character                    | String Edit Descriptors         |
| 6.1.2                             | Structure Components            |
| 11.3.3.3 Data                     | Structures                      |
| 2.4.3.1.3 Constant                | Subobject                       |
| 14.7.1 Definition of Objects and  | Subobjects                      |
| 12.5.2.2 Function                 | Subprogram                      |
| 12.5.2.3 Subroutine               | Subprogram                      |
| 12.5.2.4 Instances of a           | Subprogram                      |
| 11.2 External                     | Subprograms                     |
| 12.5.2 Procedures Defined by      | Subprograms                     |
| Effects of INTENT Attribute on    | Subprograms 12.5.2.1            |
| 13.9.3 Bit Copy                   | Subroutine                      |
| 13.2.2 Elemental Intrinsic        | Subroutine Arguments            |
| 12.4.4                            | Subroutine Reference            |

|                                   |                                   |
|-----------------------------------|-----------------------------------|
| 12.4.5 Elemental Intrinsic        | Subroutine Reference              |
| 12.5.2.3                          | Subroutine Subprogram             |
| 13.11 Intrinsic                   | Subroutines                       |
| 13.9 Intrinsic                    | Subroutines                       |
| 13.9.1 Date and Time              | Subroutines                       |
| 6.2.2.3.2 Vector                  | Subscript                         |
| 6.2.2.3.1                         | Subscript Triplet                 |
| 6.1.1                             | Substrings                        |
| 14.5 Scope of the Assignment      | Symbol                            |
| 2.1 High Level                    | Syntax                            |
| 3.2 Low-Level                     | Syntax                            |
| Characteristics 1.5.3             | Syntax Conventions and            |
| 1.5.1                             | Syntax Rules                      |
| 1.5.2 Assumed                     | Syntax Rules                      |
| 10.6.1.1                          | T, TL, and TR Editing             |
| 5.1.2.8                           | TARGET Attribute                  |
| 5.2.8                             | TARGET Statement                  |
| 6.3.1.2 Allocation of Pointer     | Targets                           |
| 6.3.3.2 Deallocation of Pointer   | Targets                           |
| 8.1.4.4.4 Loop                    | Termination                       |
| Statements 9.4.6                  | Termination of Data Transfer      |
| 2.5 Fundamental                   | Terms                             |
| 2 Fortran                         | Terms and Concepts                |
| 3.4 Including Source              | Text                              |
| 1.5.4                             | Text Conventions                  |
| 10.6.1.1 T,                       | TL, and TR Editing                |
| 3 Characters, Lexical             | Tokens, and Source Form           |
| 10.6.1.1 T, TL, and               | TR Editing                        |
| File Position Prior to Data       | Transfer 9.2.1.3.2                |
| File Position after Data          | Transfer 9.2.1.3.3                |
| 9.4.4.1 Direction of Data         | Transfer                          |
| 9.4.4.4 Data                      | Transfer                          |
| 9.4.4.4.1 Unformatted Data        | Transfer                          |
| 9.4.4.4.2 Formatted Data          | Transfer                          |
| 13.10.11                          | Transfer Function                 |
| 13.6                              | Transfer Function                 |
| 9.4.2 Data                        | Transfer Input/Output List        |
| 9.4.4 Execution of a Data         | Transfer Input/Output Statement   |
| 9.4 Data                          | Transfer Statements               |
| 9.4.6 Termination of Data         | Transfer Statements               |
| 6.2.2.3.1 Subscript               | Triplet                           |
| 2.4.1 Data                        | Type                              |
| 2.4.1.1 Intrinsic                 | Type                              |
| 2.4.1.2 Derived                   | Type                              |
| 4.1 The Concept of Data           | Type                              |
| 4.3.1.1 Integer                   | Type                              |
| 4.3.1.2 Real                      | Type                              |
| 4.3.1.3 Complex                   | Type                              |
| 4.3.2.1 Character                 | Type                              |
| 4.3.2.2 Logical                   | Type                              |
| 5.1.1.7 Derived                   | Type                              |
| 5.1                               | Type Declaration Statements       |
| Primary 7.1.4.1 Data Type,        | Type Parameters, and Shape of a   |
| Expression 7.1.4 Data Type,       | Type Parameters, and Shape of an  |
| Result of an/ 7.1.4.2 Data Type,  | Type Parameters, and Shape of the |
| 5.1.1                             | Type Specifiers                   |
| of a Primary 7.1.4.1 Data         | Type, Type Parameters, and Shape  |
| of an Expression 7.1.4 Data       | Type, Type Parameters, and Shape  |
| of the Result of an/ 7.1.4.2 Data | Type, Type Parameters, and Shape  |
| 4 Intrinsic and Derived Data      | Types                             |
| 4.3 Intrinsic Data                | Types                             |
| 4.3.1 Numeric                     | Types                             |
| 4.3.2 Nonnumeric                  | Types                             |
| 4.4 Derived                       | Types                             |
| 4.4.2 Determination of Derived    | Types                             |
| 4.2 Relationship of               | Types and Values to Objects       |
| References 14.1.2.3               | Unambiguous Generic Procedure     |
| 7.3.1                             | Unary Defined Operation           |
| Variables That Are Initially      | Undefined 14.7.4                  |
| That Cause Variables to Become    | Undefined 14.7.6 Events           |
| 14.7 Definition and               | Undefined of Variables            |
| 3.1.3                             | Underscore                        |
| 9.4.4.4.1                         | Unformatted Data Transfer         |
| 9.1.2                             | Unformatted Record                |
| INQUIRE Statement 9.6.1.12        | UNFORMATTED= Specifier in the     |
| 9.3.2 Connection of a File to a   | Unit                              |
| 9.4.4.2 Identifying a             | Unit                              |
| 2.2 Program                       | Unit Concepts                     |
| 9.3.1                             | Unit Existence                    |
| 11 Program                        | Units                             |
| 11.4 Block Data Program           | Units                             |

## Permuted Index for Headings

Scope of External Input/Output Units 14.3  
11.3.2 The USE Statement and Use Association  
12.1.2.2.2 Host Association and USE Association  
Association 14.6.1.2 Use Association and Host  
6 Use of Data Objects  
11.3.3 Examples of the Use of Modules  
11.3.2 The USE Statement and Use Association  
2.4.2 Data Value  
10.8.1.1 Null Values  
10.9.1.2 Namelist Input Values  
10.9.1.4 Null Values  
4.1.1 Set of Values  
4.4.3 Derived-Type Values  
Construction of Derived-Type Values 4.4.4  
4.5 Construction of Array Values  
4.2 Relationship of Types and Values to Objects  
2.4.3.1.1 Variable  
Definition and Undefined of Variables 14.7  
14.7.2 Variables That Are Always Defined  
Defined 14.7.3 Variables That Are Initially  
Undefined 14.7.4 Variables That Are Initially  
14.7.5 Events That Cause Variables to Become Defined  
14.7.6 Events That Cause Variables to Become Undefined  
Functions 13.8.3 Vector and Matrix Multiplication  
Functions 13.10.13 Vector and Matrix Multiply  
6.2.2.3.2 Vector Subscript  
7.5.3 Masked Array Assignment WHERE  
6.2.1 Whole Arrays  
Statement 9.6.1.19 WRITE= Specifier in the INQUIRE  
10.6.1.2 X Editing

## ANNEX F. INDEX

(This Annex is not part of this standard, but is included for information only.)

- accessibility attribute 5-5
- access-id* R522 5-10
- access-spec* R510 5-5
- access-stmt* R521 5-10
- ac-do-variable* R435 4-13
- ac-IMPLIED-DO* R433 4-13
- ac-IMPLIED-DO-CONTROL* R434 4-13
- action-stmt* R216 2-2
- action-term-do-construct* R827 8-6
- active 8-7
- actual-arg* R1213 12-8
- actual-arg-spec* R1211 12-8
- ac-value* R432 4-12
- add-op* R710 3-3
- add-op* R710 7-2
- add-operand* R706 7-2
- advancing input/output statement 9-3
- allocatable array 5-7
- ALLOCATABLE attribute 5-9
- allocatable-stmt* R526 5-11
- ALLOCATE statement 6-6
- allocate-object* R625 6-6
- allocate-stmt* R622 6-6
- allocation* R624 6-6
- alphanumeric-character* R302 3-1
- alt-return-spec* R1214 12-8
- and-op* R720 3-3
- and-op* R720 7-3
- and-operand* R715 7-3
- approximation methods 4-4
- argument keyword 2-8
- arithmetic-if-stmt* R840 8-13
- array 2-7
- array 6-3
- array constructor 4-12
- array element 2-7
- array element order 6-4
- array elements 6-3
- array intrinsic assignment statement 7-18
- array pointer 5-7
- array section 2-7
- array section 6-4
- array-constructor* ~~R431~~ 4-12
- array-element* R615 ~~6-3~~
- array-section* R616 6-3
- array-spec* R512 5-6
- ASCII collating sequence 4-7
- assigned-goto-stmt* R839 8-12
- assignment-stmt* R735 7-18
- assign-stmt* R838 8-12
- Association 2-9
- assumed type parameter 5-3
- assumed-shape array 5-7
- assumed-shape-spec* R516 5-7
- assumed-size array 5-8
- assumed-size-spec* R518 5-8
- attributes 5-1
- attr-spec* R503 5-1
- automatic array 5-6
- automatic data object 5-2
- backspace-stmt* R919 9-18
- belongs 8-1
- binary-constant* R408 4-3
- blank common 5-18
- blank-interp-edit-desc* R1015 10-3
- block 8-1
- block* R801 8-1
- block-data* R1110 11-6
- block-data* R1110 2-1
- block-data-stmt* R1111 11-6
- block-do-construct* R817 8-6
- boz-literal-constant* R407 4-3
- branch target statement 8-11
- Branching 8-11
- c* R1017 10-3
- call-stmt* R1210 12-8
- CASE construct 8-3
- case index 8-4
- case-construct* R808 8-3
- case-expr* R812 8-3
- case-selector* R813 8-3
- case-stmt* R810 8-3
- case-value* R815 8-3
- case-value-range* R814 8-3
- character constant expression 7-7
- character context 3-4
- character intrinsic assignment statement 7-18
- character intrinsic operation 7-5
- character literal constant 4-5
- character* R301 3-1
- character relational intrinsic operation 7-5
- character sequence structure 5-4
- character sequence type 4-9
- character storage unit 14-7
- character string 4-5
- character string edit descriptor 10-2
- character type 4-5
- characteristics of a procedure 12-2
- char-constant* R309 3-3
- char-expr* R726 7-6
- char-initialization-expr* R731 7-8
- char-length* R508 5-4
- char-literal-constant* R420 4-6
- char-selector* R506 5-4
- char-string-edit-desc* R1016 10-3
- char-variable* R605 6-1
- CLOSE statement 9-9
- close-spec* R908 9-9
- close-stmt* R907 9-9
- collating sequence 4-6

## Index

- comment 3-5
- comment 3-6
- common block storage sequence 5-19
- common blocks 5-18
- COMMON statement 5-18
- common-block-object* R549 5-18
- common-stmt* R548 5-18
- complex type 4-5
- complex-literal-constant* R417 4-5
- component-array-spec* R428 4-8
- component-attr-spec* R427 4-8
- component-decl* R429 4-8
- component-def-stmt* R426 4-8
- components 4-1
- computed-goto-stmt* R837 8-12
- concatenation 4-6
- concat-op* R712 3-3
- concat-op* R712 7-3
- conformable 2-8
- connected 9-5
- connect-spec* R905 9-6
- constant 2-7
- constant expression 7-7
- constant* R305 3-2
- constant subobject 2-7
- constant-subobject* R702 7-1
- CONTAINS statement 12-15
- contains-stmt* R1225 12-15
- continue-stmt* R841 8-13
- Control characters 3-1
- control edit descriptor 10-2
- control information list 9-10
- control-edit-desc* R1010 10-2
- create a file 9-2
- current record 9-3
- currently allocated 6-7
- cycle-stmt* R834 8-8
- d* R1008 10-2
- data edit descriptor 10-2
- data entity 2-6
- data object 2-7
- data object reference 2-9
- DATA statement 5-12
- data transfer input statement 9-1
- data transfer output statements 9-1
- data type 2-6
- data type 4-1
- data-edit-desc* R1005 10-2
- data-i-do-object* R536 5-12
- data-i-do-variable* R537 5-13
- data-implied-do* R535 5-12
- data-ref* R612 6-2
- data-stmt* R529 5-12
- data-stmt-constant* R533 5-12
- data-stmt-object* R531 5-12
- data-stmt-repeat* R534 5-12
- data-stmt-set* R530 5-12
- data-stmt-value* R532 5-12
- DEALLOCATE statement 6-7
- deallocate-stmt* R628 6-7
- declaration 2-8
- declaration-construct* R207 2-1
- default character 4-5
- default complex 4-5
- default integer 4-2
- default logical 4-7
- default real 4-4
- default-char-expr* R727 7-6
- default-char-variable* R606 6-1
- default-int-variable* R608 6-1
- default-logical-variable* R604 6-1
- deferred-shape array 5-7
- deferred-shape-spec* R517 5-7
- defined 2-8
- defined assignment statement 7-19
- defined binary operation 7-5
- defined operation 7-5
- defined unary operation 7-5
- defined-binary-op* R724 3-3
- defined-binary-op* R724 7-4
- defined-operator* R311 3-3
- defined-unary-op* R704 3-3
- defined-unary-op* R704 7-2
- definition 2-8
- definition 2-9
- delete a file 9-2
- deleted features 1-5
- Delimiters 3-4
- derived type 2-6
- derived-type intrinsic assignment statement 7-18
- derived-type-def* R422 4-8
- derived-type-stmt* R424 4-8
- digits 3-1
- digit-string* R402 4-3
- DIMENSION attribute 5-6
- dimension-stmt* R525 5-11
- direct access 9-3
- direct access input/output statement 9-12
- disassociated 2-8
- DO termination 8-7
- do-block* R823 8-6
- do-body* R828 8-6
- do-construct* R816 8-5
- do-stmt* R818 8-6
- do-term-action-stmt* R829 8-6
- do-term-shared-stmt* R833 8-7
- double precision real 4-4
- do-variable* R822 8-6
- dummy procedure 12-2
- dummy-arg* R1221 12-13
- e* R1009 10-2
- edit descriptor 10-2
- element sequence 12-10
- elemental 12-1



- elemental function 13-1
- elemental reference 12-10
- elemental subroutine 13-1
- else-if-stmt* R804 8-2
- else-stmt* R805 8-2
- elsewhere-stmt* R742 7-22
- END statement 2-5
- end-block-data-stmt* R1112 11-6
- end-do* R824 8-6
- end-do-stmt* R825 8-6
- endfile record 9-1
- endfile-stmt* R920 9-18
- end-function-stmt* R1218 12-11
- end-if-stmt* R806 8-2
- ending point 6-2
- end-interface-stmt* R1203 12-4
- end-module-stmt* R1106 11-2
- end-of-file condition 9-15
- end-of-record condition 9-15
- end-program-stmt* R1103 11-1
- end-select-stmt* R811 8-3
- end-subroutine-stmt* R1222 12-13
- end-type-stmt* R425 4-8
- end-where-stmt* R743 7-22
- entity-decl* R504 5-1
- ENTRY statement 12-13
- entry-stmt* R1223 12-13
- EQUIVALENCE statement 5-17
- equivalence-object* R547 5-17
- equivalence-set* R546 5-17
- equivalence-stmt* R545 5-17
- equiv-op* R722 3-3
- equiv-op* R722 7-4
- equiv-operand* R717 7-3
- executable program 2-3
- executable statement 2-4
- executable-construct* R215 2-2
- executable-program* R201 2-1
- execution cycle 8-8
- execution-part* R208 2-2
- execution-part-construct* R209 2-2
- exist 9-2
- exit-stmt* R835 8-8
- explicit 12-3
- explicit-shape array 5-6
- explicit-shape-spec* R513 5-6
- exponent* R416 4-4
- exponent-letter* R415 4-4
- expr* R723 7-4
- expression 2-7
- expression 7-1
- extended-intrinsic-op* R312 3-4
- extension operation 7-5
- extension operator 7-5
- extent 2-7
- EXTERNAL attribute 5-9
- external file 9-2
- external procedure 12-1
- external procedure 2-4
- EXTERNAL statement 12-7
- external subprogram 2-3
- external unit 9-5
- external-file-unit* R902 9-5
- external-stmt* R1207 12-7
- external-subprogram* R203 2-1
- field 10-3
- field width 10-3
- file 9-2
- file connection statements 9-1
- file inquiry statement 9-1
- file positioning statements 9-1
- file-name-expr* R906 9-7
- fixed source form 3-6
- format control 10-3
- format* R913 9-11
- format-item* R1003 10-2
- format-specification* R1002 10-1
- format-stmt* R1001 10-1
- formatted input/output statement 9-11
- formatted record 9-1
- Fortran character set 3-1
- free source form 3-4
- function 2-3
- function reference 2-7
- function subprogram 12-11
- function-reference* R1209 12-8
- function-stmt* R1216 12-11
- function-subprogram* R1215 12-11
- function-subprogram* R1215 2-1
- generic identifier 12-5
- generic interface 12-5
- generic name 12-5
- generic-spec* R1206 12-4
- global entity 14-1
- goto-stmt* R836 8-12
- Graphic characters 3-1
- hex-constant* R410 4-3
- hex-digit* R411 4-3
- host 11-2
- host 2-4
- host association 12-1
- host scoping unit 2-3
- IF construct 8-1
- IF statement 8-1
- if-construct* R802 8-2
- if-stmt* R807 8-3
- if-then-stmt* R803 8-2
- imaginary part 4-5
- imag-part* R419 4-5
- implicit 12-3
- IMPLICIT statement 5-14
- implicit-part* R205 2-1
- implicit-part-stmt* R206 2-1
- implicit-spec* R541 5-14

## Index

- implicit-stmt* R540 5-14
- inactive 8-7
- INCLUDE line 3-6
- initial point 9-3
- initialization expression 7-8
- initialization-expr* R730 7-8
- inner-shared-do-construct* R832 8-7
- Input statements 9-1
- input-item* R914 9-13
- inquire by file 9-19
- inquire by output list 9-19
- inquire by unit 9-19
- inquire-spec* R924 9-20
- inquire-stmt* R923 9-20
- inquiry function 13-1
- instance 12-13
- int-constant* R308 3-3
- integer constant expression 7-7
- integer type 4-2
- INTENT attribute 5-5
- intent-spec* R511 5-5
- intent-stmt* R519 5-9
- interface 12-3
- interface body 12-4
- interface-block* R1201 12-4
- interface-body* R1204 12-4
- interface-stmt* R1202 12-4
- internal procedure 12-1
- internal procedure 2-4
- internal subprogram 2-3
- internal unit 9-5
- internal-file-unit* R903 9-5
- internal-subprogram* R211 2-2
- internal-subprogram-part* R210 2-2
- int-expr* R728 7-6
- int-initialization-expr* R732 7-8
- int-literal-constant* R404 4-3
- intrinsic 2-9
- intrinsic assignment statement 7-18
- INTRINSIC attribute 5-9
- intrinsic binary operation 7-4
- intrinsic function 13-1
- intrinsic operation 7-4
- intrinsic procedure 12-1
- INTRINSIC statement 12-7
- intrinsic type 2-6
- intrinsic unary operation 7-4
- intrinsic-operator* R310 3-3
- intrinsic-stmt* R1208 12-7
- int-variable* R607 6-1
- io-control-spec* R912 9-10
- io-implied-do* R916 9-13
- io-implied-do-control* R918 9-14
- io-implied-do-object* R917 9-13
- io-unit* R901 9-5
- iteration count 8-7
- k R1011 10-2
- keyword 2-8
- keyword* R1212 12-8
- kind 4-2
- kind 4-4
- kind 4-5
- kind 4-5
- kind 4-7
- kind type parameter 2-6
- kind-param* R405 4-3
- kind-selector* R505 5-1
- label R313 3-4
- label-do-stmt* R819 8-6
- left tab limit 10-10
- length 4-5
- length-selector* R507 5-4
- letters 3-1
- letter-spec* R542 5-14
- level-1-expr* R703 7-2
- level-2-expr* R707 7-2
- level-3-expr* R711 7-3
- level-4-expr* R713 7-3
- level-5-expr* R718 7-3
- Lexical tokens 3-2
- line 3-4
- list-directed input/output statement 9-12
- literal constant 2-7
- literal-constant* R306 3-2
- local entity 14-1
- logical constant expression 7-7
- logical intrinsic assignment statement 7-18
- logical intrinsic operation 7-5
- logical type 4-7
- logical-expr* R725 7-6
- logical-initialization-expr* R733 7-8
- logical-literal-constant* R421 4-7
- logical-variable* R603 6-1
- loop 8-5
- loop-control* R821 8-6
- lower-bound* R514 5-6
- low-level syntax 3-2
- m R1007 10-2
- main program 11-1
- main-program* R1101 11-1
- main-program* R1101 2-1
- many-one array section 6-5
- masked array assignment 7-21
- mask-expr* R741 7-22
- module 2-4
- module procedure 12-1
- module procedure 2-4
- module* R1104 11-2
- module* R1104 2-1
- module reference 11-2
- module reference 2-9
- module subprogram 2-3
- module-procedure-stmt* R1205 12-4
- module-stmt* R1105 11-2

- module-subprogram* R213 2-2
- module-subprogram-part* R212 2-2
- mult-op* R709 3-3
- mult-op* R709 7-2
- mult-operand* R705 7-2
- n* R1013 10-2
- name 2-8
- name association 14-5
- name* R304 3-2
- named common blocks 5-18
- named constant 2-7
- named file 9-2
- named-constant* R307 3-3
- named-constant-def* R539 5-14
- namelist input/output statement 9-12
- NAMELIST statement 5-16
- namelist-group-object* R544 5-16
- namelist-stmt* R543 5-16
- Names 3-2
- name-value subsequences 10-15
- next effective item 9-16
- next record 9-3
- nonadvancing input/output statement 9-3
- nonblock-do-construct* R826 8-6
- nonexecutable statement 2-4
- nonlabel-do-stmt* R820 8-6
- nonnumeric types 4-5
- not-op* R719 3-3
- not-op* R719 7-3
- NULLIFY statement 6-7
- nullify-stmt* R626 6-7
- numeric constant expression 7-7
- numeric intrinsic assignment statement 7-18
- numeric intrinsic operation 7-4
- numeric intrinsic operator 7-4
- numeric relational intrinsic operation 7-5
- numeric sequence structure 5-4
- numeric sequence type 4-9
- numeric storage unit 14-6
- numeric types 4-2
- numeric-expr* R729 7-6
- object 2-7
- obsolescent features 1-5
- octal-constant* R409 4-3
- only* R1109 11-2
- OPEN statement 9-6
- open-stmt* R904 9-6
- operator 2-9
- OPTIONAL attribute 5-9
- optional-stmt* R520 5-10
- or-op* R721 3-3
- or-op* R721 7-3
- or-operand* R716 7-3
- outer-shared-do-construct* R830 8-6
- Output statements 9-1
- output-item* R915 9-13
- PARAMETER attribute 5-5
- PARAMETER statement 5-14
- parameter-stmt* R538 5-14
- parent object 6-2
- parent-string* R610 6-2
- partially associated 14-7
- part-ref* R613 6-2
- pause-stmt* R844 8-13
- pointer 2-8
- pointer association status 14-6
- POINTER attribute 5-9
- pointer-assignment-stmt* R736 7-21
- pointer-object* R627 6-7
- pointer-stmt* R527 5-11
- position 9-2
- position-edit-desc* R1012 10-2
- position-spec* R922 9-19
- power-op* R708 3-3
- power-op* R708 7-2
- preceding record 9-3
- Preconnection 9-6
- prefix* R1217 12-11
- present 12-15
- primary* R701 7-1
- PRINT statement 9-10
- printing 9-18
- print-stmt* R911 9-10
- PRIVATE 5-10
- private-sequence-stmt* R423 4-8
- procedure 2-3
- procedure interface block 2-4
- procedure reference 2-9
- processor 1-1
- processor dependent 1-2
- program name 11-1
- program unit 2-3
- program-stmt* R1102 11-1
- program-unit* R202 2-1
- PUBLIC 5-10
- r* R1004 10-2
- range 8-7
- range 8-7
- rank 2-7
- rank 2-7
- READ statement 9-10
- reading 9-1
- read-stmt* R909 9-10
- real part 4-5
- real type 4-4
- real-literal-constant* R413 4-4
- real-part* R418 4-5
- record 9-1
- record number 9-3
- RECURSIVE 12-12
- reference 6-1
- relational intrinsic operation 7-5
- rel-op* R714 3-3
- rel-op* R714 7-3

## Index

- rename* R1108 11-2
- repeat specification 10-2
- representable character 4-6
- representation method 4-4
- representation methods 4-2
- representation methods 4-5
- representation methods 4-7
- restricted expression 7-8
- result variable 2-3
- RETURN statement 12-14
- return-stmt* R1224 12-14
- rewind-stmt* R921 9-18
- SAVE attribute 5-8
- saved object 5-8
- saved-entity* R524 5-10
- save-stmt* R523 5-10
- scalar 2-7
- scalar 6-2
- scale factor 10-3
- scope 14-1
- scoping unit 2-3
- section-subscript* R618 6-3
- select-case-stmt* R809 8-3
- sequence 2-9
- SEQUENCE statement 4-9
- sequence structure 5-4
- sequence type 4-8
- sequential access 9-2
- sequential access input/output statement 9-12
- set of allowed access methods 9-2
- set of allowed actions 9-2
- set of allowed forms 9-2
- set of allowed record lengths 9-2
- shape 2-8
- shape conformance 7-7
- share 8-7
- shared-term-do-construct* R831 8-6
- sign* R406 4-3
- signed-digit-string* R401 4-2
- signed-int-literal-constant* R403 4-3
- sign-edit-desc* R1014 10-3
- signed-real-literal-constant* R412 4-4
- significand* R414 4-4
- size 2-8
- size of a common block 5-19
- size of a storage sequence 14-6
- source forms 3-4
- special characters 3-1
- specific interface 12-4
- Specific names 13-1
- specification expression 7-9
- specification-expr* R734 7-9
- specification-part* R204 2-1
- specification-stmt* R214 2-2
- standard module 1-5
- standard-conforming program 1-1
- starting point 6-2
- statement 3-4
- statement entity 14-1
- statement function 12-2
- statement keyword 2-8
- statement label 8-12
- stat-variable* R623 6-6
- stmt-function-stmt* R1226 12-17
- stop-code* R843 8-13
- stop-stmt* R842 8-13
- storage associated 14-7
- Storage association 14-6
- storage sequence 14-6
- storage unit 14-6
- stride 6-5
- stride* R620 6-3
- structure 2-6
- structure 5-4
- structure constructor 4-12
- structure-component* R614 6-2
- structure-constructor* R430 4-12
- Subobject 2-7
- subobject designator 2-8
- subobject* R602 6-1
- subroutine 2-3
- subroutine subprogram 12-12
- subroutine-stmt* R1220 12-12
- subroutine-subprogram* R1219 12-12
- subroutine-subprogram* R1219 2-1
- subscript* R617 6-3
- subscript-triplet* R619 6-3
- substring 6-2
- substring* R609 6-2
- substring-range* R611 6-2
- Syntax rules 1-3
- TARGET attribute 5-9
- target* R737 7-21
- target-stmt* R528 5-12
- terminal point 9-3
- totally associated 14-7
- transformational functions 13-1
- type declaration statement 5-1
- type parameter 2-6
- type specifier 5-3
- type-declaration-stmt* R501 5-1
- type-param-value* R509 5-4
- type-spec* R502 5-1
- undefined 2-8
- underscore* R303 3-1
- unformatted input/output statement 9-11
- unformatted record 9-1
- unit 9-5
- unspecified storage unit 14-7
- upper-bound* R515 5-6
- use associated 11-2
- Use association 14-5
- USE statement 11-2
- use-stmt* R1107 11-2

value separator 10-13  
variable 2-7  
*variable* R601 6-1  
vector subscript 6-5  
*vector-subscript* R621 6-3  
*w* R1006 10-2  
*where-construct* R739 7-21  
*where-construct-stmt* R740 7-21  
*where-stmt* R738 7-21  
whole array 6-3  
whole array named constant 6-3  
whole array variable 6-3  
WRITE statement 9-10  
*write-stmt* R910 9-10  
writing 9-1



