X3J3 / S8.111
March 1989

Fortran 8x

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

American National Standard for Information Systems Programming Language

Fortran

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

Secretariat: Computer and Business Equipment Manufacturers Association

Draft S8, Version 111
Submitted to X3 by X3J3, American National Standards Institute, Inc.

FOREWORD

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

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

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

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

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

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

OVERVIEW

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

(1) Array operations

(2) Improved facilities for numerical computation

(3) Parameterized nonnumeric intrinsic data types

(4) User-defined data types

(5) Pointers

(6) Facilities for modular data and procedure definitions

(7) The concept of language evolution

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

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

The FORTRAN 77 arithmetic, logical, and character operations and intrinsic (predefined) functions are extended to operate on array-valued operands. These include whole, partial, and masked array assignment, array-valued constants and expressions, and facilities to define user-supplied array-valued functions. New intrinsic functions are provided to manipulate and construct arrays,

to perform gather/scatter operations, and to support extended computational capabilities involving arrays. (For example, an intrinsic function is provided to sum the elements of an array.)

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

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

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

Procedure definitions may be used to define operations on intrinsic or derived data types and nonintrinsic assignments for intrinsic and derived types.

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

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

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

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

DOCUMENT ORGANIZATION

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

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

High/Low Level Concepts. Section 2 (Fortran Terms and Concepts) contains many of the high-level concepts of Fortran. This includes the concept of an executable program and the relationships of its major parts. Also included are the syntax of program units, the rules for statement ordering, and the definition of many of the fundamental terms used throughout the document.

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

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

Section 4 (Intrinsic and Derived Data Types) describes the distinction between a data type and a data object, and then focuses on data type. It defines a data type as a set of data values, corresponding forms (constants) for representing these values, and operations on these values. The concept of an intrinsic data type is introduced, and the properties of Fortran's intrinsic types (INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, and CHARACTER) are described. Note that only type concepts are described here, and not the declaration and properties of data objects.

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

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

Section 6 (Use of Data Objects) deals mainly with the concept of a variable, and describes the various forms that variables may take. Scalar variables include character strings and substrings, structured (derived-type) objects, structure components, and array elements. Arrays are considered to be variables, as are array sections. Among the array facilities described here are array sections (subarrays), and array allocation and deallocation (user controlled dynamic arrays). The section concludes with a summary of the allowed appearances of array names.

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

Section 7 also describes assignment of computational results to data objects, which has three principal forms: the conventional assignment statement, the pointer assignment statement, and the WHERE statement and construct. The WHERE statement and construct allow masked array assignment.

Section 13 (Intrinsic Procedures) describes more than one hundred intrinsic procedures that provide a rich set of computational capabilities. In addition to the FORTRAN 77 intrinsic functions, this includes many array processing functions, a comprehensive set of numerical environmental inquiry functions, and a set of procedures for manipulation of bits in nonnegative integer data.

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

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

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

Program Units. Section 11 (Program Units) describes main programs, modules, and block data program units. Modules, along with the USE statement, are described as a mechanism for encapsulating data and procedure definitions that are to be used by (accessible to) other program units. Modules are described as vehicles for defining global derived-type definitions, global data object declarations, procedure libraries, and combinations thereof.

Section 12 (Procedures) contains a comprehensive treatment of procedure definition and invocation, including that for user-defined functions and subroutines. The concepts of implicit and explicit procedure interfaces are explained, and situations requiring explicit procedure interfaces are identified. The rules governing actual and dummy arguments, and their association, are described.

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

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

defined (have predictable values) and events that cause variables to become undefined.

X3J3 COMMITTEE

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

STEERING COMMITTEE

Jeanne C. Adams, Chair
Jerrold L. Wagener, Vice-Chair
Walter S. Brainerd, Director, Technical Work
Lloyd W. Campbell, Editor
John K. Reid, Secretary
Jeanne T. Martin, Convenor, ISO/IECJTC1/SC22/WG5
Neldon H. Marshall, Librarian
E. Andrew Johnson, Interpretations and International Representative
Kurt W. Hirchert, Vocabulary Representative

SUBGROUP HEADS	(Assistant Heads)
Richard A. Hendrickson	(Brian T. Smith)
Kurt W. Hirchert	(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)

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

Jeanne C. Adams, Chair
Jerrold L. Wagener, Vice-Chair
Martin N. Greenfield, Vice-Chair (1972-1985)
Walter S. Brainerd, Director, Technical Work*
Lloyd W. Campbell, Editor*
John K. Reid, Secretary
Jeanne T. Martin, Secretary* (1982-1987), Convener ISO/TC97/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. Hirchert, Vocabulary Representative*
James H. Matheny, Vocabulary Representative* (1986-1987)

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

*Subgroup Head

TABLE OF CONTENTS

FOREWORD i

1. INTRODUCTION 1-1

1.1 Purpose 1-1
1.2 Processor 1-1
1.3 Scope 1-1
1.3.1 Inclusions 1-1
1.3.2 Exclusions 1-1
1.4 Conformance 1-1
1.5 Notation Used in This Standard 1-2
1.5.1 Syntax Rules 1-2
1.5.2 Assumed Syntax Rules 1-3
1.5.3 Syntax Conventions and Characteristics 1-3
1.5.4 Text Conventions 1-4
1.6 Deleted and Obsolescent Features 1-4
1.6.1 Nature of Deleted Features 1-4
1.6.2 Nature of Obsolescent Features 1-4
1.7 Modules 1-4

2. FORTRAN TERMS AND CONCEPTS 2-1

2.1 High Level Syntax 2-1
2.2 Program Unit Concepts 2-3
2.2.1 Scoping Unit 2-3
2.2.2 Executable Program 2-3
2.2.3 Main Program 2-3
2.2.4 Procedure 2-3
2.2.5 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 Constant 2-7
2.4.5 Variable 2-7
2.4.6 Scalar 2-7
2.4.7 Array 2-7
2.4.8 Pointer 2-8
2.4.9 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

3. CHARACTERS, LEXICAL TOKENS, AND SOURCE FORM 3-1

3.1 Fortran 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 Character Graphics 3-2
3.1.6 Representable Characters 3-2
3.1.7 Collating Sequence 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 3-3
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 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 and Entities 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 4-6
4.4.1 Derived-Type Definition 4-7
4.4.2 Determination of Derived Types 4-9
4.4.3 Derived-Type Values 4-9
4.4.4 Construction of Derived-Type Values 4-9
4.4.5 Derived-Type Operations and Assignment 4-10
4.5 Construction of Array Values 4-10

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 5-4
5.2 Attribute Specification Statements 5-8
5.2.1 INTENT Statement 5-9
5.2.2 OPTIONAL Statement 5-9
5.2.3 Accessibility Statements 5-9
5.2.4 SAVE Statement 5-10
5.2.5 DIMENSION Statement 5-10
5.2.6 DATA Statement 5-10
5.2.7 PARAMETER Statement 5-12
5.3 IMPLICIT Statement 5-13
5.4 NAMELIST Statement 5-14
5.5 Storage Association of Data Objects 5-15
5.5.1 EQUIVALENCE Statement 5-15

5.5.2 COMMON Statement 5-17

6. USE OF DATA OBJECTS 6-1

6.1 Scalars 6-1
6.1.1 Substrings 6-1
6.1.2 Structure Components 6-2
6.2 Arrays 6-2
6.2.1 Whole Arrays 6-3
6.2.2 Array Elements and Array Sections 6-3
6.3 Dynamic Association 6-5
6.3.1 ALLOCATE Statement 6-6
6.3.2 NULLIFY Statement 6-6
6.3.3 DEALLOCATE Statement 6-7
6.3.4 Summary of Array Name Appearances 6-8

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 7-12
7.2.1 Numeric Intrinsic Operations 7-12
7.2.2 Character Intrinsic Operation 7-13
7.2.3 Relational Intrinsic Operations 7-13
7.2.4 Logical Intrinsic Operations 7-14
7.3 Interpretation of Defined Operations 7-15
7.3.1 Unary Defined Operation 7-15
7.3.2 Binary Defined Operation 7-15
7.4 Precedence of Operators 7-16
7.5 Assignment 7-17
7.5.1 Assignment Statement 7-17
7.5.2 Pointer Assignment Statement 7-20
7.5.3 Masked Array Assignment—WHERE 7-20

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 8-11
8.2.2 GO TO Statement 8-11
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 8-12
8.3 CONTINUE Statement 8-12
8.4 STOP Statement 8-13

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 9-8
9.4 Data Transfer Statements 9-9
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 9-14
9.4.4 Execution of a Data Transfer Input/Output Statement 9-15
9.4.5 Printing of Formatted Records 9-17
9.4.6 Termination of Data Transfer Statements 9-17
9.5 File Positioning Statements 9-18
9.5.1 BACKSPACE Statement 9-18
9.5.2 ENDFILE Statement 9-18
9.5.3 REWIND Statement 9-19
9.6 File Inquiry 9-19
9.6.1 Inquiry Specifiers 9-19
9.6.2 Restrictions on Inquiry Specifiers 9-22
9.6.3 IOLENGTH= Specifier in the INQUIRE Statement 9-22
9.7 Restrictions on Function References and List Items 9-22
9.8 Restriction on Input/Output Statements 9-22

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 10-8
10.5.3 Character Editing 10-8
10.5.4 Generalized Editing 10-8
10.6 Control Edit Descriptors 10-9
10.6.1 Position Editing 10-9
10.6.2 Slash Editing 10-10
10.6.3 Colon Editing 10-10
10.6.4 S, SP, and SS Editing 10-10

10.6.5 P Editing 10-10
10.6.6 BN and BZ Editing 10-11
10.7 Character String Edit Descriptors 10-11
10.7.1 Character Constant Edit Descriptor 10-11
10.7.2 H Editing 10-11
10.8 List-Directed Formatting 10-11
10.8.1 List-Directed Input 10-12
10.8.2 List-Directed Output 10-13
10.9 Namelist Formatting 10-14
10.9.1 Namelist Input 10-14
10.9.2 Namelist Output 10-16

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 11-1
11.2 Procedures 11-2
11.2.1 Internal Procedures 11-2
11.2.2 Host Association 11-2
11.3 Modules 11-2
11.3.1 Module Reference 11-2
11.3.2 The USE Statement 11-3
11.3.3 Examples of the Use of Modules 11-4
11.4 Block Data Program Units 11-5

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 12-1
12.2.1 Characteristics of Dummy Arguments 12-1
12.2.2 Characteristics of Function Results 12-2
12.3 Procedure Interface 12-2
12.3.1 Implicit and Explicit Interfaces 12-2
12.3.2 Specification of the Procedure Interface 12-2
12.4 Procedure Reference 12-6
12.4.1 Actual Argument List 12-6
12.4.2 Function Reference 12-8
12.4.3 Elemental Intrinsic Function Reference 12-9
12.4.4 Subroutine Reference 12-9
12.4.5 Elemental Subroutine Reference 12-9
12.5 Procedure Definition 12-9
12.5.1 Intrinsic Procedure Definition 12-9
12.5.2 Procedures Defined by Subprograms 12-9
12.5.3 Definition of Procedures by Means Other Than Fortran 12-14
12.5.4 Statement Function 12-14
12.5.5 Overloading Names 12-15

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
13.4 Argument Presence Inquiry Functions 13-1
13.5 Numeric, Mathematical, Character, 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 Inquiry Functions 13-2
13.5.6 Logical Functions 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 13-4
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 13-5
13.8.8 Array Manipulation Functions 13-5
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 13-6
13.10.2 Numeric Functions 13-6
13.10.3 Mathematical Functions 13-6
13.10.4 Character Functions 13-6
13.10.5 Character Inquiry Functions 13-7
13.10.6 Kind Inquiry Function 13-7
13.10.7 Logical Functions 13-7
13.10.8 Numeric Inquiry Functions 13-7
13.10.9 Bit Inquiry Functions 13-7
13.10.10 Bit Manipulation Functions 13-7
13.10.11 Transfer Function 13-8
13.10.12 Floating-point Manipulation Functions 13-8
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 13-9
13.10.17 Array Reshape Function 13-9
13.10.18 Array Manipulation Functions 13-9
13.10.19 Array Location Functions 13-9
13.10.20 Pointer Association Status Inquiry Functions 13-9
13.11 Intrinsic Subroutines 13-9
13.12 Specific Names for Intrinsic Functions 13-10

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 14-2
14.2 Scope of Labels 14-3
14.3 Scope of External Input/Output Units 14-3
14.4 Scope of Operators 14-3
14.5 Scope of the Assignment Symbol 14-3
14.6 Association 14-3
14.6.1 Name Association 14-3
14.6.2 Pointer Association 14-4
14.6.3 Storage Association 14-4
14.7 Definition and Undefinition of Variables 14-6
14.7.1 Definition of Objects and Subobjects 14-6
14.7.2 Variables That Are Always Defined 14-6
14.7.3 Variables That Are Initially Defined 14-6
14.7.4 Variables That Are Initially Undefined 14-6
14.7.5 Events That Cause Variables to Become Defined 14-6
14.7.6 Events That Cause Variables to Become Undefined 14-7

A. FORTRAN FAMILY OF STANDARDS A-1

A.1 The Fortran Language Standard A-1
A.1.1 Primary Features A-1
A.1.2 Incremental Features A-1
A.1.3 Decremental Features A-1
A.1.4 Compatibility A-1
A.1.5 Core A-2
A.2 Supplementary Standards Based on Procedure Libraries A-2
A.2.1 Interface Mechanisms A-2
A.3 Supplementary Standards Based on Module Libraries A-2
A.3.1 Interface Mechanisms A-3
A.4 Rules for Supplementary Standards A-4
A.5 Secondary Standards A-5
A.6 Standard Conformance A-5
A.6.1 Name Registration A-5
A.7 Fortran Family of Standards A-5

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

C. SECTION NOTES C-1

C.1 Section 1 Notes C-1
C.1.1 Conformance (1.4) C-1
C.1.2 Obsolescence (1.6.2) C-1

C.2 Section 2 Notes C-1
C.3 Section 3 Notes C-1
C.3.1 Collating Sequence (3.1.7) 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 Intrinsic and Derived Data Types (4.3, 4.4) C-2
C.4.3 Selection of the Approximation Methods C-3
C.4.4 Components and Storage of Derived Types (4.4.1) C-3
C.4.5 Pointers C-3
C.4.6 The POINTER Attribute (5.1.2.7) C-4
C.5 Section 5 Notes C-5
C.5.1 Type Declaration Statements (5.1) C-5
C.5.2 The TARGET Attribute C-5
C.6 Section 6 Notes C-5
C.6.1 Substrings (6.1.1) C-5
C.6.2 Array Element References (6.2.2) C-5
C.6.3 Structure Components (6.1.2) C-6
C.6.4 Pointer Allocation and Association C-6
C.7 Section 7 Notes C-7
C.7.1 Character Assignment C-7
C.7.2 Evaluation of Function References C-7
C.7.3 Pointers in Expressions C-7
C.7.4 Pointers on the Left Side of an Assignment C-7
C.8 Section 8 Notes C-8
C.8.1 Loop Control C-8
C.8.2 The CASE Construct C-8
C.8.3 Examples of Invalid DO Constructs C-8
C.9 Section 9 Notes C-9
C.9.1 Input/Output Records (9.1) C-9
C.9.2 Files (9.2) C-9
C.9.3 OPEN Statement (9.3.4) C-10
C.9.4 Connection Properties (9.3.2) C-11
C.9.5 CLOSE Statement (9.3.5) C-12
C.9.6 INQUIRE Statement C-13
C.9.7 Keyword Specifiers C-13
C.9.8 Format Specifications (9.4.1.1) C-13
C.9.9 Unformatted Input/Output (9.4.4.4.1) C-14
C.9.10 Input/Output Restrictions C-14
C.9.11 Pointers in an Input/Output List C-14
C.10 Section 10 Notes C-14
C.10.1 Character Constant Format Specification (10.1.2, 10.7.1) C-14
C.10.2 T Edit Descriptor (10.6.1.1) C-14
C.10.3 Length of Formatted Records C-14
C.10.4 Number of Records (10.3, 10.4, 10.6.2) C-14
C.10.5 List-Directed Input/Output (10.8) C-15
C.10.6 List-Directed Input (10.8.1) C-15
C.11 Section 11 Notes C-15
C.11.1 Main Program and Block Data Program Unit (11.1, 11.4) C-16
C.11.2 Dependent Compilation (11.3) C-16
C.11.3 Pointers in Modules C-17
C.11.4 Example of a Module (11.3) C-18

C.12 Section 12 Notes C-20
C.12.1 External Procedures (12.3.2.2) C-20
C.12.2 Procedures Defined by Means Other Than Fortran (12.5.3) C-21
C.12.3 Procedure Interfaces (12.3) C-21
C.12.4 Argument Association (12.4.1) C-21
C.12.5 Argument Intent Specification (12.4.1.1) C-22
C.12.6 Dummy Argument Restrictions (12.5.2.9) C-22
C.12.7 Pointers as Arguments C-23
C.12.8 The ASSOCIATED Function C-23
C.12.9 Internal Procedure Restrictions C-23
C.12.10 The Result Variable (12.5.2.2) C-23
C.13 Section 13 Notes C-23
C.13.1 Summary of Features C-23
C.13.2 Examples C-25
C.13.3 FORmula TRANslation and Array Processing C-28
C.13.4 Sum of Squared Residuals C-29
C.13.5 Vector Norms: Infinity-Norm and One-Norm C-30
C.13.6 Matrix Norms: Infinity-Norm and One-Norm C-30
C.13.7 Logical Queries C-30
C.13.8 Parallel Computations C-30
C.13.9 Example of Element-by-Element Computation C-31
C.14 Section 14 Notes C-31
C.14.1 Storage Association of Zero-Sized Objects C-31

F. GLOSSARY OF TECHNICAL TERMS F-1

FIGURES

Figure 2.1. Requirements on Statement Ordering. 2-5
Figure A.1. The Fortran Language Standard. A-2
Figure A.2. Supplementary Standards. A-3
Figure A.3. Secondary Standards. A-4
Figure A.4. The Fortran Family of Standards. A-6

TABLES

Table 2.1. Statements Allowed in Scoping Units. 2-5
Table 6.1. Subscript Order Value. 6-4
Table 6.2. Allowed Appearances of Array Names. 6-8
Table 7.1. Type of Operands and Result for the Intrinsic Operation $[x_1]$ *op* x_2 7-5
Table 7.2. Interpretation of the Numeric Intrinsic Operators. 7-12
Table 7.3. Interpretation of the Character Intrinsic Operator //. 7-13
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 Assignment Statement *variable = expr* 7-18
Table 7.9. Numeric Conversion and Assignment Statement *variable = expr*. 7-18
Table 14.2. Summary Comparison of Use and Host Associations 14-4
Table C.1. Values Assigned to INQUIRE Specifier Variables. C-13

1. INTRODUCTION

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

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

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

1.3.1 Inclusions. This standard specifies:

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

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

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

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

1.3.2 Exclusions. This standard does not specify:

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

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

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

(4) The program and processor behavior when the rules of this standard fail to establish an interpretation except for the processor detection and reporting requirements in items (2), (3), and (4) of 1.4

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

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

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

(8) The physical properties and implementation of storage

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

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

A processor conforms to this standard if:

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

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

(3) It contains the capability to detect and report the use within a submitted program unit of an additional form or relationship that is not permitted by the numbered syntax rules or their associated constraints.

(4) It contains the capability to detect and report the use within a submitted program of kind type parameter values (4.3) not supported by the processor.

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

A standard-conforming processor may allow additional forms and relationships provided that such additions do not conflict with the standard forms and relationships. However, a standard-conforming processor may allow additional intrinsic procedures even though this could cause a conflict with the name of a procedure in a standard-conforming program. If such a conflict occurs and involves the name of an external procedure, the processor is permitted to use the intrinsic procedure unless the name appears in an EXTERNAL statement in the same scoping unit (2.2.1). A standard-conforming program must not use nonstandard intrinsic procedures that have been added by the processor.

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

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

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

1.5 Notation Used In This Standard.

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

1.5.1 Syntax Rules.

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

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

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

Some common abbreviations used in syntactic terms are:

stmt for statement *attr* for attribute

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

(3) The syntactic metasymbols used are:

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

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

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

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

An example of the use of syntax rules is:

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

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

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

When specific entities are substituted for *digit*, actual integer literal constants might be:

```
4
67
1999
10243852
```

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

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

xyz-name **is** *name*

scalar-xyz **is** *xyz*

Constraint: *scalar-xyz* must be scalar.

1.5.3 Syntax Conventions and Characteristics.

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

considered to be a source form statement is given a "*-stmt*" ending in the syntax rules.

(2) The rules on statement ordering are described rigorously in the definition of *program-unit* (R202-R215). Expression hierarchy is described rigorously in the definition of *expr* (R723).

(3) The suffix "*-spec*" is used consistently for specifiers, such as keyword type parameters, keyword actual arguments, and input/output statement specifiers. It also is used for type declaration attribute specifications (for example, "*array-spec*" in R512), and in a few other ad hoc cases.

(4) When reference is made to a type parameter, including the surrounding parentheses, the term "selector" is used. See, for example, "*length-selector*" (R507) and "*kind-selector*" (R505).

(5) The term "*subscript*" (e.g., R614, R615, and R618) is used consistently in array definitions.

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

1.6 Deleted and Obsolescent Features.

This standard protects the users' investment in existing software by including all of the language elements of ANSI X3.9-1978. This document identifies two categories of outmoded features. There are none in the first category, **deleted features**, which consists of features considered to have been redundant in ANSI X3.9-1978 and largely unused. Those in the second category, **obsolescent features**, are considered to have been redundant in ANSI X3.9-1978, but are still used frequently.

1.6.1 Nature of Deleted Features.

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

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

1.6.2 Nature of Obsolescent Features.

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

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

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

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

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

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

1.7 Modules.

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

A module may be standardized as a separate collateral standard. A **standard module** must not use any deleted or obsolescent features, nor any nonstandard form or relations.

2. FORTRAN TERMS AND CONCEPTS

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

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

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

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

R1101 *main-program* **is** [*program-stmt*]
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-program-stmt

R1217 *external-subprogram* **is** *procedure-heading*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
procedure-ending

R1218 *procedure-heading* **is** *function-stmt*
or *subroutine-stmt*

R1220 *procedure-ending* **is** *end-function-stmt*
or *end-subroutine-stmt*

R1104 *module* **is** *module-stmt*
[*specification-part*]
[*module-subprogram-part*]
end-module-stmt

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

R203 *specification-part* **is** [*use-stmt*] ...
[*implicit-part*]
[*declaration-construct*] ...

R204 *implicit-part* **is** [*implicit-part-stmt*] ...
implicit-stmt

R205 *implicit-part-stmt* **is** *implicit-stmt*
or *parameter-stmt*
or *format-stmt*
or *entry-stmt*

R206 *declaration-construct* **is** *derived-type-def*
or *interface-block*
or *type-declaration-stmt*
or *specification-stmt*
or *parameter-stmt*
or *format-stmt*
or *entry-stmt*

			or *stmt-function-stmt*
R207	*execution-part*		**is** *executable-construct* [*execution-part-construct*] ...
R208	*execution-part-construct*		**is** *executable-construct* **or** *format-stmt* **or** *data-stmt* **or** *entry-stmt*
R209	*internal-subprogram-part*		**is** *contains-stmt* *internal-subprogram* [*internal-subprogram*] ...
R210	*internal-subprogram*		**is** *procedure-heading* [*specification-part*] [*execution-part*] *procedure-ending*
R211	*module-subprogram-part*		**is** *contains-stmt* *module-subprogram* [*module-subprogram*] ...
R212	*module-subprogram*		**is** *procedure-heading* [*specification-part*] [*execution-part*] [*internal-subprogram-part*] *procedure-ending*
R213	*specification-stmt*		**is** *access-stmt* **or** *common-stmt* **or** *data-stmt* **or** *dimension-stmt* **or** *equivalence-stmt* **or** *external-stmt* **or** *intent-stmt* **or** *intrinsic-stmt* **or** *namelist-stmt* **or** *optional-stmt* **or** *save-stmt*
R214	*executable-construct*		**is** *action-stmt* **or** *case-construct* **or** *do-construct* **or** *if-construct* **or** *where-construct*
R215	*action-stmt*		**is** *allocate-stmt* **or** *assignment-stmt* **or** *backspace-stmt* **or** *call-stmt* **or** *close-stmt* **or** *computed-goto-stmt* **or** *continue-stmt* **or** *cycle-stmt* **or** *deallocate-stmt* **or** *endfile-stmt* **or** *end-function-stmt* **or** *end-program-stmt* **or** *end-subroutine-stmt* **or** *exit-stmt* **or** *goto-stmt*

or *if-stmt*
or *inquire-stmt*
or *nullify-stmt*
or *open-stmt*
or *pointer-assignment-stmt*
or *print-stmt*
or *read-stmt*
or *return-stmt*
or *rewind-stmt*
or *stop-stmt*
or *where-stmt*
or *write-stmt*
or arithmetic-if-stmt
or assign-stmt
or assigned-goto-stmt
or pause-stmt

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

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

(1) A derived-type definition (4.4.1),

(2) A procedure interface block, excluding any procedure interface blocks contained within it (12.3.2.1), or

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

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

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

2.2.3 Main Program. The main program is described in 11.1.

2.2.4 Procedure. A **procedure** encapsulates arbitrary computations that may be invoked directly during program execution. A principal difference between the two kinds of procedures is the way in which each is invoked. A **function** is a procedure that is invoked in an expression; its invocation causes a value to be computed which is then used in evaluating the expression. A **subroutine** is a procedure that is invoked in a CALL statement or by a defined assignment statement (12.4.4, 12.3.2.1). A subroutine may be used to change the program state by changing the values of any of the data objects accessible to the subroutine; a function may do this in addition to computing the function value.

Procedures are described further in Section 12.

2.2.4.1 External Procedure. An **external procedure** is a procedure that is defined by an external subprogram or by means other than Fortran. An external procedure may be invoked by the main program or any procedure of an executable program.

2.2.4.2 Module Procedure. A **module procedure** is a procedure that is defined by a module subprogram (R212). A module procedure may be invoked by another module subprogram in the module or by any program unit using the module. The module containing the subprogram is called the **host** of the module procedure.

2.2.4.3 Internal Procedure. An **internal procedure** is a procedure that is defined by an internal subprogram (R210). The containing main program or subprogram is called the **host** of the internal procedure. An internal procedure is local to its host in the sense that the internal procedure is accessible within the scoping units of the host and all its other internal procedures but is not accessible elsewhere.

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

2.2.5 Module. A **module** contains (or accesses from other modules) definitions that are to be made accessible to other program units. These definitions include data object declarations, type definitions, procedure definitions, and procedure interface blocks. The purpose of a module is to make the definitions it contains accessible to all other program units in an executable program that request such accessibility. A scoping unit in another program unit may request access to the definitions contained in a module. Modules are further described in Section 11.

2.3 Execution Concepts. A program unit is a sequence of statements. Each statement is classified as either an **executable statement** or a **nonexecutable statement**. There are restrictions on the order in which statements may appear in a program unit, and certain executable statements may appear only in certain executable constructs.

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

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

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

Figure 2.1 Requirements on Statement Ordering.

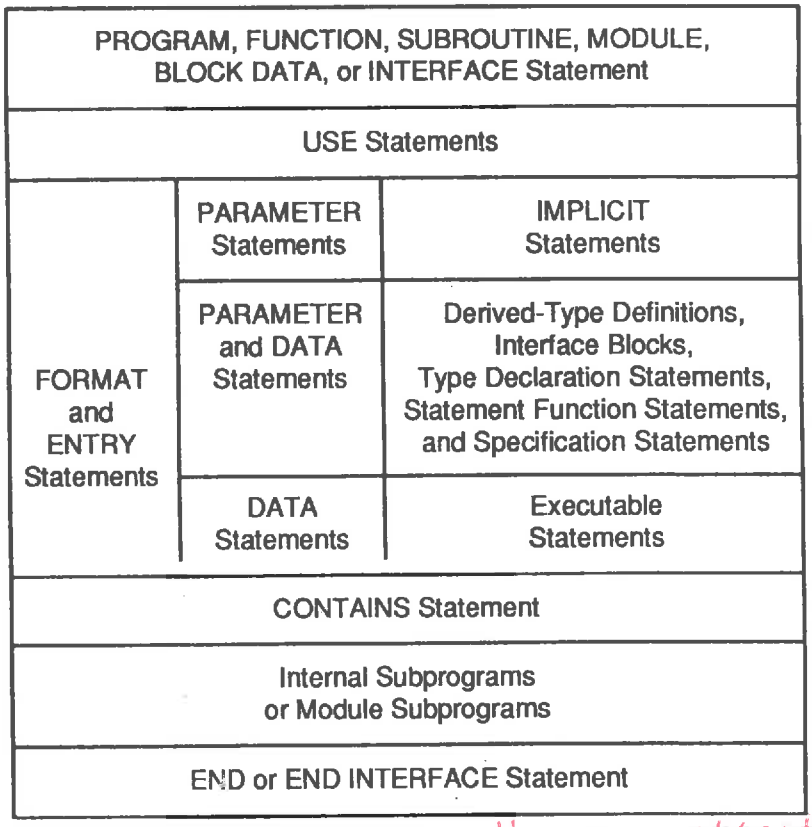


Table 2.1 Statements Allowed in Scoping Units.

Kind of program unit:	Main Program	Module	Block Data	External Subprog	Module Subprog	Internal Subprog	Interface Block
USE Statement	Yes	Yes	No	Yes	Yes	Yes	Yes
ENTRY Statement	No	No	No	Yes	Yes	No	No
FORMAT Statement	Yes	No	No	Yes	Yes	Yes	No
Misc. Declarations (See Note)	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Derived-Type Definition	Yes	Yes	No	Yes	Yes	Yes	Yes
Interface 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

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

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

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

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

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

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

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

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

2.4 Data Concepts.

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

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

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

2.4.1.1 Intrinsic Type. An **intrinsic type** is a type that is implicitly defined, along with operations, and is always accessible. The intrinsic types are INTEGER, REAL, COMPLEX, CHARACTER, and LOGICAL. The properties of intrinsic types are described in 4.3. An intrinsic type may be parameterized, in which case the set of data values depends on the values of the parameters. Such a parameter is called a **type parameter**.

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

A derived-type definition is local to the scoping unit in which it appears, but may be accessed from other scoping units by use or host association (14.6.1.2).

Derived types are described further in 4.4.

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

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

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

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

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

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

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

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

2.4.6 Scalar. A **scalar** is a datum that is not an array. Scalars may be of any intrinsic type or derived type. Note that a structure is scalar even if it has arrays as components.

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

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

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

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

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

2.4.8 Pointer. A **pointer** is a data object that has the POINTER attribute. It must not be referenced or defined until it is associated with a target by allocation (6.3.1) or pointer assignment (7.5.2). A pointer is **disassociated** if it not currently associated with a target. If it is an array, the rank is declared, but the extents are determined when the pointer is associated with a target. If associated, a pointer may appear as a primary in an expression anywhere a variable with the same type, type parameters, and shape may appear. The value of the primary is the value of the target.

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

There are two categories of physical **storage units**: numeric and character. When used in a storage association context, scalar objects of type default integer, default real, and default logical each use a single numeric storage unit. When used in a storage association context, scalar objects of type double precision real and default complex each use two contiguous numeric storage units. When used in a storage association context, each character in an object of type default character uses one character storage unit and scalar default character objects employ a contiguous set of such units. When used in a storage association context, array objects are assigned contiguous storage units of the appropriate category, in array element order (6.2.2.2). For example, the storage order for a two-dimensional array is the first column followed by the second column, etc.

Objects having different categories of storage units must not be storage associated. Derived-type objects and objects having a kind type-parameter value other than the default or double precision must not appear in a storage association context.

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

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

keywords in I/O context?

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

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

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

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

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

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

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

2.5.6 Association. An **association** exists if an entity may be identified by different names in the same scoping unit or by the same name or different names in different scoping units. It may be name association (14.6.1), pointer association (14.6.2), or storage association (14.6.3). Name association may be argument association, host association, or use association.

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

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

3. CHARACTERS, LEXICAL TOKENS, AND SOURCE FORM

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

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

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

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

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

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

The twenty-six letters define the syntactic class *letter*. If a processor also permits lower-case letters, the lower-case letters are equivalent to the corresponding upper-case letters in program units except in character constants, character constant edit descriptors, and H edit descriptors.

— what about character constants in I/O control ?

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

0 1 2 3 4 5 6 7 8 9

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

3.1.3 Underscore.

R303 *underscore* **is** _

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

3.1.4 Special Characters. The twenty-one **special characters** are:

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

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

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

3.1.6 Representable Characters. Additional characters may be representable in the processor, but may appear only in character constants, character string edit descriptors and comments (4.3.2.1, 10.2.1, 3.3.1.3, 3.3.2.1). They define the syntactic class *rep-char*.

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

ICHAR ('*character*')

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

The only constraints on the collating sequence are:

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

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

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

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

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

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

Note that the intrinsic functions ACHAR and IACHAR provide conversions between the characters and the integers according to the **ASCII collating sequence** as specified in ANS X3.4-1977. Note also that the intrinsic functions LGT, LGE, LLE, and LLT provide comparisons between strings based on the ASCII collating sequence.

3.2 Low-Level Syntax.

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

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

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

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

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

Examples of names:

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

TRAILER_ (trailing underscore)

3.2.3 Constants.

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

R306 *literal-constant* **is** *int-literal-constant*
or *real-literal-constant*
or *complex-literal-constant*
or *logical-literal-constant*
or *char-literal-constant*
or *boz-literal-constant*

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

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

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

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

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

3.2.4 Operators.

R310 *intrinsic-operator* **is** *power-op*
or *mult-op*
or *add-op*
or *concat-op*
or *rel-op*
or *not-op*
or *and-op*
or *or-op*
or *equiv-op*

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

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

R710 *add-op* **is** +
or -

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

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

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

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

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

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

or .NEQV.

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

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

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

R312 *overloaded-intrinsic-op* **is** *intrinsic-operator*

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

3.2.5 Statement Labels. A statement label provides a means of referring to an individual statement. Any statement not forming part of another statement may be labeled.

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

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

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

```
99999
10
 010
```

are all statement labels. The last two are equivalent.

3.2.6 Delimiters. A **delimiter** is one of a pair of characters used to enclose character constants or other lexical tokens. The following pairs are delimiters:

```
( ... )
/ ... /
" ... "
' ... '
. ... .
(/ ... /)
```

3.3 Source Form. A Fortran program unit is an ordered sequence of one or more Fortran statements and comments. A Fortran **statement** is an ordered sequence of one or more complete or partial lines. A **line** is an ordered sequence of zero or more characters. Lines following a program unit END statement are not part of that program unit.

Any syntax rule ending in "*-stmt*" identifies a Fortran statement.

A **character context** means characters within (between the delimiters of) a character literal constant or within a character string edit descriptor.

Any character representable in the processor may occur in a character context or in a comment.

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

3.3.1 Free Source Form. In **free source form**, each source line may contain from zero to 132 characters and there are no restrictions on where a statement may appear within a line. However, if any statement on the line contains any character that is not of default kind, the number of characters allowed on the line is processor dependent.

In free form, blank characters must not appear within lexical tokens. Blanks may be inserted freely between tokens to improve readability. A sequence of blank characters outside of a character context is equivalent to a single blank character.

A blank must be used to separate names, constants, or labels from adjacent keywords, names, constants, or labels. For example, in

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

the blanks are required after REAL, READ, 30, and DO.

One or more blanks must be used to separate some adjacent keywords and may be optionally used within other keywords.

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

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

3.3.1.2 Free Form Statement Separation. The character ";" separates statements, or partial statements, on a single source line except when it appears in a character context or in a comment. If the sequence delimited by one or more ";" separators contains no characters or only blank characters, the sequence is ignored.

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

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

first character position of the next noncomment line.

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

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

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

3.3.2 Fixed Source Form. In **fixed source form**, each line must contain exactly 72 characters and there are restrictions on where a statement may appear within a line.

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

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

3.3.2.2 Fixed Form Statement Separation. The character ";" separates statements, or partial statements, on a single source line except when it appears in a character context or in a comment. If the sequence delimited by one or more ";" separators contains no characters or only blank characters, the sequence is ignored.

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

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

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

INCLUDE *char-literal-constant*

An INCLUDE line is not a Fortran statement.

An INCLUDE must appear on a single source line where a statement may appear; it must be the only nonblank text on this line other than an optional trailing comment. Thus, a statement label is not allowed. The source text of a program unit may contain a processor-dependent number of

INCLUDE lines. minimum 2?

The effect of the INCLUDE line is as if the referenced source text physically replaces the INCLUDE line prior to program processing. Included text may contain any source text, including additional INCLUDE lines; such nested INCLUDE lines are similaraly replaced with the specified source text. The maximum depth of nesting of any nested INCLUDE lines is processor dependent. minimum 2

However, when an INCLUDE line or a set of nested INCLUDE lines is resolved, neither the first included statement line nor the last must represent part of an incomplete statement present in the containing program unit before any INCLUDE line is processed.

The interpretation of *char-literal-constant* is processor dependent. An example of a possible valid interpretation is that *char-literal-constant* is the name of a file that contains the source text to be included. See cr 17:8-9

4. INTRINSIC AND DERIVED DATA TYPES

Fortran provides an abstract means that permits the categorization of data without relying on a particular physical representation. This abstract means is the concept of **data type**. Each data type has a name. The names of the intrinsic types are defined by the language; the names of any derived types must be defined in type definitions (4.4.1). A data type is characterized by a set of values, a means to denote the values, and a set of operations that can manipulate and interpret the values.

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

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

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

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

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

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

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

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

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

A constant value may be given a name (5.1.2.1, 5.2.7).

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

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

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

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

Scalar data of any intrinsic or derived type may be shaped in a rectangular pattern to compose an array. An array is an object and has a type just as a scalar object does. Thus data objects, such as arrays, may be collections of subobjects.

A scalar object of derived type is referred to as a structure. The components of a structure are subobjects.

Variables may be objects or subobjects. The data type of a variable determines which values that variable may take. Assignment provides one means of defining or redefining the value of a variable of any type. Assignment is defined intrinsically for all types when the type, type parameters, and shape of both the variable and the value to be assigned to it are identical. Assignment between objects of certain differing intrinsic types, type parameters, and shapes is described in Section 7. For example, assignment of an integer value to a real variable is defined intrinsically. For an assignment that is not defined intrinsically, conversions may be defined by a subroutine (7.5.1.3) with the ASSIGNMENT interface (12.3.2.1).

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

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

4.3 Intrinsic Data Types. The intrinsic data types are:

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

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

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

4.3.1.1 Integer Type. The set of values for the integer type is a subset of the mathematical integers. A processor must provide one or more **representation methods** that define sets of values for data of type integer. Each such method is characterized by a value for a type parameter called the **kind** type parameter. The kind type parameter of a representation method is returned by the intrinsic inquiry function KIND (13.13.51). The decimal exponent range of a representation method is returned by the intrinsic function RANGE (13.13.85). The kind type parameter value must specify a representation method that exists on the processor.

If two methods have kind type parameters k_1 and k_2 with $k_1 < k_2$, the decimal range of the first is less than or equal to the decimal range of the second.

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

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

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

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

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

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

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

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

R406 *sign* **is** +
or –

Examples of unsigned and signed integer literal constants are:

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

where SHORT is a scalar integer named constant.

An integer constant is interpreted as a decimal value.

In a DATA statement (5.2.6), an unsigned binary, octal, or hexadecimal literal constant must correspond to an integer scalar entity.

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

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

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

Constraint: *digit* may have only the values 0 or 1.

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

Constraint: *digit* may have only the values 0 through 7.

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

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

In these constants, the binary, octal, and hexadecimal digits are interpreted according to their respective number systems. If the processor supports lower-case letters, a lower-case letter as a hexadecimal digit is interpreted as an upper-case letter.

4.3.1.2 Real and Double Precision Real Type. The real type has values that approximate the mathematical real numbers. A processor must provide two or more **approximation methods** that define sets of values for data of type real. Each such method is characterized by a value for a type parameter called the **kind** type parameter. The kind type parameter of an approximation method is returned by the intrinsic inquiry function KIND (13.13.51). The decimal precision and decimal exponent range of an approximation method are returned by the intrinsic functions PRECISION (13.13.79) and RANGE (13.13.85). The kind type parameter value must specify an approximation method that exists on the processor.

If two methods have kind type parameters k_1 and k_2 with $k_1 < k_2$, the decimal precision of the first is less than or equal to the decimal precision of the second.

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

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

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

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

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

R415 *exponent* **is** *signed-digit-string*

R416 *exponent-letter* **is** E
or D

A real literal constant written without an exponent part, or with exponent letter E and without the optional kind type parameter, is a default real object; exponent letter D specifies a double precision real constant. A real literal constant written with a kind type parameter is a real constant with the specified kind type parameter.

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

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

Examples of signed real literal constants are:

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

In the ~~second~~ sixth example (10.93E7_QUAD), the named integer constant QUAD must have been defined and its value must be a kind type parameter value for the real type.

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

Any approximation method used to represent data entities of type real may be used for both the real and imaginary parts of a data entity of type complex. A kind type parameter may be specified for a complex entity and selects one real approximation method for both parts following the same rules as for the real type.

If a kind type parameter is not specified, the type of both parts is default real and the complex data entity is **default complex**. There is no keyword for default double precision complex.

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

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

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

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

If the real part and 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 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 parameter value.

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 type parameter value of the complex literal constant is that of the signed real literal constant.

Examples of complex literal constants are:

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

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

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

A processor 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. The kind type parameter value must specify a representation method that exists on the processor.

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

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

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

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

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

A *rep-char* is any character in the set of values representable for the type character with kind *kind-param* if present, and of type default character otherwise. The delimiting apostrophes or quotation marks are not part of the value of the character literal constant.

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

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

Examples of character literal constants are:

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

both of which have the value DON'T and

```
''
```

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

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

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

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

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

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

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

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

4.4 Derived Types. Additional data types may be derived from the intrinsic data types. A type definition is required to define the name of the type and the names and types of its components. A nonpointer component may be declared to be of any intrinsic or previously defined derived type. A pointer component may be a pointer to an object of intrinsic type or to an object of any derived type defined in the containing scoping unit, including the type being defined. Ultimately, a derived type is resolved into a sequence of components of intrinsic type or pointers.

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

4.4.1 Derived-Type Definition.

R422 *derived-type-def* **is** *derived-type-stmt*
[PRIVATE]
[SEQUENCE]
component-def-stmt
[*component-def-stmt*]...
end-type-stmt

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

Constraint: If SEQUENCE is present, all derived types used in component definitions must also be SEQUENCE structures.

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

R424 *end-type-stmt* **is** END TYPE [*type-name*]

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

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

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

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

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

Constraint: A *type-spec* (5.1) in a *component-def-stmt* may include the *type-name* of its containing *derived-type-def* only if the POINTER attribute is specified for that component.

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

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

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

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

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

Constraint: A *char-length* in a *component-decl* must be an integer constant expression.

Constraint: Each bound in the *explicit-shape-spec* (R425 and R426) must be an integer constant expression.

If the SEQUENCE keyword is present, the structure is a sequenced structure. If all of the components ultimately all resolve to entities of default numeric or default logical type, the structure is a numeric storage associated structure. If all of the components ultimately resolve to entities of default character type, the structure is a character storage associated structure. With some restrictions, an entity with a sequenced type can appear in COMMON and EQUIVALENCE statements and can have its type independently declared in different scoping units. The processor must allocate storage for these entities in the manner and order it would have if the individual components had been directly enumerated in a common block.

Note that the double colon separator in a *component-def-stmt* is required only if the DIMENSION attribute, the POINTER attribute, or both are specified; otherwise, it is optional.

A component is array valued if its *component-decl* contains an *explicit-shape-spec-list* (5.1.2.4) or its *component-def-stmt* contains the DIMENSION attribute. If the *component-decl* contains an *explicit-shape-spec-list*, it specifies the array bounds; otherwise, the *explicit-shape-spec-list* in the DIMENSION attribute specifies the array bounds.

If a component of a derived type is of a type declared to be private, either all components of the derived type must be private or the derived type must be private. A type definition is PRIVATE if the *derived-type-stmt* includes a PRIVATE *access-spec* or if the default accessibility (5.2.3) is PRIVATE and the *derived-type-stmt* does not include a PUBLIC *access-spec*. If a type definition is PRIVATE, then the type name, the structure value constructor (4.4.4) for the type, any entity that is of the type, and any procedure that has a dummy argument that is of the type are accessible only within the module containing the definition.

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

An accessible name of a derived-type component may be preceded by a structure name of the same derived type and the % character to select that component of the structure (6.1.2.). Note that a component may be an array; when selected by component name qualification, such an object is an array even though the parent object may be a scalar object.

An example of a derived-type definition is:

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

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

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

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

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

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

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

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

An example of a type with private components is:

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

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

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

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

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

An example of a private type is:

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

Such a type would be accessible only within the module.

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

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

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

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

```
TYPE NODE
   INTEGER                 :: VALUE
   TYPE(NODE), POINTER     :: NEXT_NODE
END TYPE
```

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

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

Two data entities have the same type if they are declared with reference to the same derived-type definition. Data entities in different scoping units have the same type if they are declared with reference to derived-type definitions which have the same name, have the SEQUENCE property, and have structure components that agree in order, name, type, and type parameters. Otherwise, they are of different derived types. If the definition of a data entity that is known in two scoping units, such as a dummy or actual argument or an entity in common, is defined with different types in the different units, the results are processor dependent.

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

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

R429 structure-constructor **is** *type-name (expr-list)*

The sequence of expressions in a derived-type constructor specifies component values that must agree in number, order, and rank with the components of the derived type. If necessary, each value is converted according to the rules of intrinsic assignment (7.1.5.4) to a value that agrees in type and type parameters with the corresponding component of the derived type. For a component that is not a pointer, the shape of the expression must agree with the shape of the component. A constructor whose component values are all constant expressions is a derived-type

constant expression.

This example uses a derived type illustrated in 4.4.1:

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

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

Where a component in the derived type is a pointer, the corresponding constructor expressions must evaluate to an object that would be an allowable target for such a pointer in a pointer assignment statement. For example, if the variable TEXT were declared (5.1) to be

```
CHARACTER, DIMENSION (1:440), TARGET :: TEXT
```

and BIBLIO were declared

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

the statement

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

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

A constant expression cannot be constructed for a derived type containing a pointer component, since a constant value is not an allowable target in a pointer assignment statement.

4.4.5 Derived-Type Operations and Assignment. Any operations on derived-type entities and nonintrinsic assignment for derived-type entities must be defined explicitly by functions or subroutines with explicit interfaces (12.3.2.1). Such definitions are described in Section 12. Arguments and function values may be of any derived or intrinsic type.

4.5 Construction of Array Values.

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

R430	*array-constructor*	**is**	*(/ ac-value-list /)*
R431	*ac-value*	**is**	*expr*
		or	*ac-implied-do*
R432	*ac-implied-do*	**is**	*(ac-value-list , ac-implied-do-control)*
R433	*ac-implied-do-control*	**is**	*ac-do-variable = scalar-int-expr ,* ■ ■ *scalar-int-expr [, scalar-int-expr]*
R434	*ac-do-variable*	**is**	*scalar-int-variable*

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

If an *ac-value* is a scalar expression, its value specifies an element of the array constructor. If an *ac-value* is an array expression, the values of the elements of the expression, in array element order, specify the corresponding sequence of elements of the array constructor. If an *ac-value* is an *ac-implied-do*, it is expanded to form a sequence of expressions, under the control of the *ac-do-variable*, as in the DO construct (8.1.4.4).

For an *ac-implied-do*, the loop initialization and execution is the same as for a DO construct. The *ac-do-variable* of an *ac-implied-do* that is contained within another *ac-implied-do* must not appear as the *ac-do-variable* of the containing *ac-implied-do*.

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

The type and type parameters of an array constructor are those of the first *ac-value*. Each *ac-value* in the sequence must have the same type and type parameters.

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

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

A one-dimensional array may be reshaped into any allowable array shape using the RESHAPE intrinsic function (13.13.88). An example is:

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

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

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

Examples of array constructors containing an implied do are:

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

and

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

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

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

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

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

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

5. DATA OBJECT DECLARATIONS AND SPECIFICATIONS

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

For example:

```
INTEGER  INCOME, EXPENDITURE
```

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

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

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

5.1 Type Declaration Statements.

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

R502 *type-spec*
is INTEGER [*kind-selector*]
or REAL [*kind-selector*]
or DOUBLE PRECISION
or COMPLEX [*kind-selector*]
or CHARACTER [*char-selector*]
or LOGICAL [*kind-selector*]
or TYPE (*type-name*)

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

R504 *entity-decl*
is *object-name* [(*array-spec*)] ■
■ [∗ *char-length*] [= *restricted-constant-expr*]
or *function-name* [(*array-spec*)] [∗ *char-length*]

R505 *kind-selector* **is** ([KIND =] *scalar-int-restricted-constant-expr*)

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

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

Constraint: The = *restricted-constant-expr* must appear if the statement contains a *PARAMETER* attribute (5.1.2.1).

Constraint: If = *restricted-constant-expr* appears, :: must appear before the *entity-decl-list*.

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

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

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

Constraint: An object must not have both the TARGET attribute and the PARAMETER attribute.

Constraint: If the POINTER attribute is specified, INTENT must not be specified.

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

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

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

Constraint: The SAVE attribute must not be specified for an object that is in a common block, a dummy argument, a procedure, a function result, or an automatic data object.

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

Constraint: An entity ~~must not~~ may have the EXTERNAL or INTRINSIC attribute ~~unless~~ only if it is a function. or = *restricted-const-expr* is present

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

An entity must not be given explicitly any of the following attributes more than once in a scoping unit: type, value, accessibility, intent, dimension, save, optional, and allocatable.

A name that identifies a specific intrinsic function in a scoping unit has a type as specified in 13.12. An explicit type declaration statement is not required; however, it is permitted. If a generic function name appears in a type declaration statement, such an appearance is not sufficient by itself to remove the generic properties from that function.

The *specification-expr* (7.1.6.2) of a *length-selector*, a *char-length*, or an *array-spec* may be a nonconstant expression provided the specification expression is in the specification part of a subprogram. If the data object being declared depends on the value of such a nonconstant expression and is not a dummy argument, such an object is called an **automatic data object**. An automatic object must not appear in a SAVE or DATA statement nor be declared with a SAVE attribute.

If a *length-selector* is a nonconstant expression, the length is declared at the entry of the procedure and is not affected by any redefinition or undefinition of the variables in the specification expression during execution of the procedure.

If an *entity-decl* contains an = *restricted-constant-expr* and the *object-name* does not have the PARAMETER attribute, *object-name* is a variable whose value is initially defined. The *object-name* becomes defined with the value determined from *restricted-constant-expr* in accordance with the rules of intrinsic assignment (7.5.1.4).

The presence of = *restricted-constant-expr* implies that *object-name* is saved. The implied SAVE attribute may be reaffirmed by explicit use of the SAVE attribute in the type declaration statement, or by inclusion of the object name in a SAVE statement (5.2.4). The = *restricted-constant-expr* must not appear if *object-name* is a dummy argument, a function result, an object in a named common block unless the type declaration is in a block data program unit, an object in blank common, and allocatable object, or an automatic object.

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

Examples of type declaration statements are:

```
LOGICAL, DIMENSION (5, 5) :: MASK1, MASK2
REAL (KIND = 10) A (10)
COMPLEX :: CUBE_ROOT = (-0.5, 0.866)
```

Examples of type declarations with kind selectors are:

```
REAL (KIND (0.0D0)) A
REAL (KIND = 2) B
COMPLEX (KIND = KIND (0.0D0)) :: C
```

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

5.1.1.1 INTEGER. The INTEGER type specifier specifies that all entities whose names are declared in this statement are of intrinsic type integer (4.3.1.1). The kind selector, if present specifies the integer representation method. If the kind selector is absent, the kind type parameter is KIND (0) and the entities declared are of type default integer.

5.1.1.2 REAL. The REAL type specifier specifies that all entities whose names are declared in this statement are of intrinsic type real (4.3.1.2). The kind selector, if present, specifies the real approximation method. If the kind selector is absent, the kind type parameter is KIND (0.0) and the entities declared are of type default real.

5.1.1.3 DOUBLE PRECISION. The DOUBLE PRECISION type specifier specifies that entities whose names are declared in this statement are of intrinsic type double precision real (4.3.1.2). The kind parameter value is KIND (0.0D0). An object declared with a type specifier REAL (KIND (0.0D0)) is of the same type as one declared with the type specifier DOUBLE PRECISION.

5.1.1.4 COMPLEX. The COMPLEX type specifier specifies that all entities whose names are declared in this statement are of intrinsic type complex (4.3.1.3). The kind selector, if present, specifies the real approximation method of the two real values making up the real and imaginary parts of the complex value. If the kind selector is absent, the kind type parameter is KIND (0.0) and the entities declared are of type default complex.

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

The length selector specifies the length of the character objects. The **char-length* may be part of an *entity-decl*, in which case the length is specified for this single entity and overrides the length specified in the length selector. If neither a length selector nor a **char-length* is specified, the length of the data entity is 1.

R506 *char-selector* **is** *length-selector*
or ([LEN=] *type-param-value* , ■
■ [KIND=] *scalar-int-restricted-constant-expr*)
or (KIND= *scalar-int-restricted-constant-expr* ■
■ [, LEN= *type-param-value*])

R507 *length-selector* **is** ([LEN =] *type-param-value*)
or * *char-length* [,]

R508 *char-length* **is** (*type-param-value*)
or *scalar-int-literal-constant*

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

R509 *type-param-value* **is** *specification-expr*
or *

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

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

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

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

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

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

Examples of character type declaration statements are:

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

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

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

5.1.1.7 Derived Type. A TYPE type specifier specifies that all entities whose names are declared in this statement are of the derived type specified by the *type-name*. The components of each such entity also are declared to be of the types specified by the corresponding *component-def* statement of the *derived-type-def* (4.4.1). When a data entity is specified to be of a derived type, the derived type must have been defined previously by a *derived-type-def*.

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

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

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

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

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

Examples of declarations with a PARAMETER attribute are:

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

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

R510 *access-spec* **is** PUBLIC
or PRIVATE

LIMITED or READONLY?

Entities that are declared with a PRIVATE attribute are not accessible outside the module. Entities that are declared with a PUBLIC attribute may be made accessible in other program units by the USE statement. The default for entities without an explicitly specified *access-spec* is PUBLIC, but this may be changed by a PRIVATE statement (5.2.3).

An example of an accessibility specification is:

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

5.1.2.3 INTENT Attribute. The **INTENT attribute** may be specified only for a dummy argument that is neither a dummy procedure, a pointer, nor an allocatable array and may appear only in the *declaration-construct* of a subprogram or interface block (12.3.2.1). An INTENT attribute specifies the intended use of the dummy argument.

R511 *intent-spec* **is** IN
or OUT
or INOUT

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

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

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

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

Dummy procedures and dummy pointers must not be declared with an INTENT attribute.

An example of an INTENT specification is:

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

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

R512 *array-spec* **is** *explicit-shape-spec-list*
or *assumed-shape-spec-list*

or *deferred-shape-spec-list*
or *assumed-size-spec*

Constraint: The maximum rank is seven.

Examples of DIMENSION attribute specifications are:

```
SUBROUTINE EX (N, A, B, S)
   REAL, DIMENSION (N, 10) :: W           ! EXPLICIT-SHAPE ARRAY
   REAL A (:), B (0:)                     ! ASSUMED-SHAPE ARRAYS
   REAL, POINTER :: D (:, :)              ! ARRAY POINTER
   REAL, DIMENSION (:), POINTER :: P      ! ARRAY POINTER
   REAL, ALLOCATABLE, DIMENSION (:) :: E ! ALLOCATABLE ARRAY
   REAL :: S (N, *)                       ! ASSUMED-SIZE ARRAY
```

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

R513 *explicit-shape-spec* **is** [*lower-bound* :] *upper-bound*

R514 *lower-bound* **is** *scalar-int-expr*

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

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

Constraint: The bounds in an explicit-shape array declaration must be specification expressions (7.1.6.2).

An **automatic array** is an explicit-shape array that is not a dummy argument and has bounds that are nonconstant expressions.

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

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

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

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

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

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

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

An **allocatable array** is a named array whose type, type parameters, name, and rank are specified in a type declaration statement containing an ALLOCATABLE attribute, but whose bounds, and hence shape, are determined when space is allocated for the array by execution of

an ALLOCATE statement (6.3.1).

An **array pointer** is an array whose type, type parameters, and rank are specified in a type declaration statement or a component definition statement, but whose bounds, and hence shape, are determined when it is associated with a target by pointer assignment (7.5.2) or execution of an ALLOCATE statement (6.3.1). An array with the pointer attribute must be declared with a *deferred-shape-list* is allocatable and may be used as a pointer or a pointer target.

R517 *deferred-shape-spec* **is** :

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

The size, bounds, and shape of an unallocated allocatable array are undefined, and no reference other than as an argument to a few intrinsic inquiry functions may be made to any part of it, nor may any part of it be defined. The lower and upper bounds of each dimension are those specified in the ALLOCATE statement when the array is allocated.

The size, bounds, and shape of the target of a disassociated array pointer are undefined. No reference may be made to any part of such an array, nor may any part of it be defined. The upper and lower bounds of each dimension are those specified in the pointer assignment statement when the array is associated or those specified in the ALLOCATE statement when the target is allocated.

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

A pointer dummy argument may be associated only with a pointer actual argument. An actual argument that is a pointer may be associated with a nonpointer dummy argument. An array-valued function may declare its result to be an array pointer.

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

R518 *assumed-size-spec* **is** [*explicit-shape-spec-list* ,] [*lower-bound* :] *

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

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

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

(2) If the actual argument associated with the assumed-size dummy array is an array element of any type other than character with a subscript order value of r (6.2.2.2) in an array of size x, the size of the dummy array is $x-r+1$.

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

The rank equals one plus the number of *explicit-shape-specs*.

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

An assumed-size array has no upper bound in its last dimension and therefore has no extent in its last dimension and no shape.

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

dimension is present, the resulting array has a shape and size.

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

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

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

Objects in the scoping unit of a module may be declared with a SAVE attribute. Such objects retain their definition status, association status, allocation status, and value when any procedure that accesses the module in a USE statement executes a RETURN or END statement. The SAVE attribute must not be specified for an object that is in a common block, a dummy argument, a procedure, a function result, or an automatic data object.

5.1.2.6 OPTIONAL Attribute. The **OPTIONAL attribute** may be specified only in the scoping unit of a subprogram or an interface block, and may be specified only for dummy arguments. The OPTIONAL attribute specifies that the dummy argument need not be associated with an actual argument in a reference to the procedure (12.5.2.8).

5.1.2.7 POINTER Attribute. The **POINTER attribute** specifies that the object must not be referenced or defined unless, as a result of executing a pointer assignment (7.5.2) or an ALLOCATE statement (6.3.1), it becomes pointer associated with a target object that may be referenced of defined. If the pointer is an array, it must be declared with a *deferred-shape-spec-list*. Examples of POINTER attribute specifications are:

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

5.1.2.8 TARGET Attribute. The **TARGET attribute** specifies that the object may have a pointer associated with it. Examples of TARGET attribute specifications are:

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

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

5.1.2.10 EXTERNAL Attribute. The **EXTERNAL attribute** specifies that an object name in a declaration containing this attribute is an external function or a dummy function and permits the name to be used as an actual argument. This attribute also may be declared via the EXTERNAL statement (12.3.2.2).

5.1.2.11 INTRINSIC Attribute. The **INTRINSIC attribute** specifies that an object name in a declaration containing this attribute is an intrinsic function and permits the name to be used as an actual argument. This attribute also may be declared via the INTRINSIC statement (12.3.2.3).

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

parameter, accessibility, intent, dimension, save, optional, external, intrinsic, pointer, target, and allocatable.

5.2.1 INTENT Statement.

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

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

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

An example of an INTENT statement is:

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

5.2.2 OPTIONAL Statement.

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

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

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

An example of an OPTIONAL statement is:

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

5.2.3 Accessibility Statements.

R521 *access-stmt* **is** *access-spec* [[::] *use-name-list*]

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

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

Constraint: A *use-name* in a PUBLIC statement must not be the name of a module procedure that has a dummy argument of PRIVATE type.

This statement declares the accessibility, **PUBLIC** or **PRIVATE**, of the entities (5.1.2.2). The default is PUBLIC.

If an *access-stmt* without a *use-name-list* appears in the scoping unit of a module, the statement sets the default accessibility that applies to all potentially accessible entities in the scoping unit of the module. The statement

```
PUBLIC
```

confirms the default of public accessibility. The statement

```
PRIVATE
```

sets the default to private accessibility.

If a PRIVATE statement appears in a *derived-type-def*, it sets the accessibility of the components of the type, overriding or confirming the accessibility of the type itself.

Examples of accessibility statements are:

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

5.2.4 SAVE Statement.

R522 *save-stmt* **is** SAVE [[::] *saved-entity-list*]

R523 *saved-entity* **is** *name*
or / *common-block-name* /

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

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

All objects named explicitly or included within a common block named explicitly have the SAVE attribute (5.1.2.5). If a particular common block name is specified in a SAVE statement in any scoping unit of an executable program other than the main program, it must be specified in a SAVE statement in every scoping unit in which that common block appears except in the scoping unit of the main program. For a common block declared in a SAVE statement, the current values of the objects in a common block storage sequence (5.5.2.1) at the time a RETURN or END statement is executed are made available to the next scoping unit in the execution sequence of the executable program that specifies the common block name or accesses the common block. If a named common block is specified in the scoping unit of the main program, the current values of the common block storage sequence are made available to each scoping unit that specifies the named common block; a SAVE statement in the scoping unit has no effect. The definition status of each object in the named common block storage sequence depends on the association that has been established for the common block storage sequence.

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

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

An example of a SAVE statement is:

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

5.2.5 DIMENSION Statement.

R524 *dimension-stmt* **is** DIMENSION *array-name* (*array-spec*) ■
■ [, *array-name* (*array-spec*)]...

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

An example of a DIMENSION statement is:

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

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

R525 *data-stmt* **is** DATA *data-stmt-set* [[,] *data-stmt-set*]...

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

A variable that appears in a DATA statement and is typed implicitly may appear in a subsequent type declaration only if that declaration confirms the implicit typing. An array name, array section, or array element that appears in a DATA statement must have had its array properties established by a previous declaration statement.

Except for variables in named common blocks, a named variable has the SAVE attribute if any part of it is initialized in a DATA statement, the named variable has the SAVE attribute, and this may be reaffirmed by a SAVE statement or a type declaration statement containing the SAVE attribute.

R526 *data-stmt-set* **is** *data-stmt-object-list / data-stmt-value-list /*

R527 *data-stmt-object* **is** *variable*
or *data-implied-do*

Constraint: The *data-stmt-object* must not be a constant.

R528 *data-stmt-value* **is** [*data-stmt-repeat* *] *data-stmt-constant*

R529 *data-stmt-constant* **is** *scalar-constant*
or *signed-int-literal-constant*
or *signed-real-literal-constant*
or *structure-constructor*
or *unsigned-boz-literal-constant*

R530 *data-stmt-repeat* **is** *scalar-int-constant*

R531 *data-implied-do* **is** (*data-i-do-object-list*, *data-i-do-variable* = ■
■ *scalar-int-expr*, *scalar-int-expr* [, *scalar-int-expr*])

R532 *data-i-do-object* **is** *array-element*
or *data-implied-do*

R533 *data-i-do-variable* **is** *scalar-int-variable*

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

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

Constraint: If a *data-stmt-constant* is a *structure-constructor*, each component must be a constant expression.

Constraint: A variable whose name is included in a *data-stmt-object-list* or a *data-i-do-object-list* must not be a dummy argument, made accessible by use association or host association, in a named common block unless the DATA statement is in a block data program unit, in a blank common block, a character string with zero length, a function name, an automatic object, a pointer, an allocatable array, or a zero-sized array.

Constraint: A subscript in an array element *data-i-do-object* must be an expression whose primaries are either constants or DO variables of the containing *data-implied-dos*. Each such DO variable must appear in some subscript of the *array-element*.

Constraint: A *scalar-int-expr* of a *data-implied-do* must involve as primaries only constants or DO variables of the containing *data-implied-dos*.

The *data-stmt-object-list* is expanded to form a sequence of scalar variables. An array whose unqualified name appears in a *data-stmt-object-list* is equivalent to a complete sequence of its array elements in array element order (6.2.2.2). An array section is equivalent to the sequence of its array elements in array element order. A *data-implied-do* is expanded to form a sequence of array elements, under the control of the implied-do DO variable, as in the DO construct (8.1.4.4).

The *data-stmt-value-list* is expanded to form a sequence of constant values. Each value must be a constant that is either previously defined or made accessible by a use association or host

association. A data statement repeat factor indicates the number of times the following constant is to be included in the sequence; omission of a data statement repeat factor has the effect of a repeat factor of one.

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

If an object is of type character or logical, the corresponding constant must be of the same type. When the object is of type integer, real, or complex, the corresponding constant must also be of type integer, real, or complex. If a constant is a binary, octal, or hexadecimal literal constant, the corresponding object must be of type integer. If an object is of derived type, the corresponding constant must be of the same type.

The value of the constant must be compatible with its corresponding variable according to the rules of intrinsic assignment (7.5.1.4), and the variable becomes initially defined with the value of the constant.

Within an interface block (12.3.2.1), data initialization by a DATA statement or by a type declaration statement has no effect.

Examples of DATA statements are:

```
CHARACTER (LEN = 10)  NAME
INTEGER, DIMENSION (0:9) :: MILES
REAL, DIMENSION (100, 100) :: SKEW
DATA  NAME / 'JOHN DOE' /, MILES / 10*0 /
DATA  ((SKEW (K, J), J = 1, K), K = 1, 100) / 5050 * 0.0 /
DATA  ((SKEW (K, J), J = K + 1, 100), K = 1, 99) / 4950 * 1.0 /
DATA MYNAME / PERSON (21, 'JOHN SMITH') /
DATA MYNAME % AGE, MYNAME % NAME / 35, 'FRED BROWN' /
```

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

5.2.7 PARAMETER Statement. The **PARAMETER statement** provides a means of defining a named constant. Named constants defined by a PARAMETER statement have exactly the same properties and restrictions as those declared in a type statement specifying a PARAMETER attribute (5.1.2.1).

R534 *parameter-stmt* **is** PARAMETER (*named-constant-def-list*)

R535 *named-constant-def* **is** *named-constant = restricted-constant-expr*

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

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

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

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

Each named constant has the PARAMETER attribute.

An example of a PARAMETER statement is:

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

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

R536 *implicit-stmt* **is** IMPLICIT *implicit-spec-list*
or IMPLICIT NONE

R537 *implicit-spec* **is** *type-spec* (*letter-spec-list*)

R538 *letter-spec* **is** *letter* [– *letter*]

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

A *letter-spec* consisting of two letters separated by a minus is equivalent to writing a list containing all of the letters in alphabetical order in the alphabetic sequence from the first letter through the second letter. For example, A–C is equivalent to A, B, C. If IMPLICIT NONE is specified, there must be no other IMPLICIT statements in the scoping unit. The same letter must not appear as a single letter, or be included in a range of letters, more than once in all of the IMPLICIT statements in a scoping unit.

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

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

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

The following are examples of the use of IMPLICIT statements:

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

SUBROUTINE SUB
   IMPLICIT COMPLEX (C)
   C = (3.0, 2.0)      ! C is implicitly declared COMPLEX
   ...
```

```
CONTAINS
   SUBROUTINE SUB1
      IMPLICIT INTEGER (A, C)
      C = (0.0, 0.0)  ! C is host associated and of
                      ! type complex
      Z = 1.0         ! Z is implicitly declared REAL
      A = 2           ! A is implicitly declared INTEGER
      CC = 1          ! CC is implicitly declared INTEGER
      ...
   END SUBROUTINE SUB1

   SUBROUTINE SUB2
      Z = 2.0         ! Z is implicitly declared REAL and
                      ! is different from the variable of
                      ! the same name in SUB1
      ...
   END SUBROUTINE SUB2

   SUBROUTINE SUB3
      USE MOD         ! Accesses the integer function FUN by use association
      Q = FUN (K)     ! Q is implicitly declared REAL and
      ...             ! K is implicitly declared INTEGER
   END SUBROUTINE SUB3
END SUBROUTINE SUB
```

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

```
INTEGER, PARAMETER :: P = SELECTED_REAL_KIND (10)
TYPE POSN
   REAL (P) X, Y
   INTEGER Z
END TYPE POSN
```

variables beginning with the letters A, B, W, X, Y, and Z are implicitly typed with the type POSN and the remaining variables are implicitly typed with type INTEGER by using the following IMPLICIT statement:

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

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

R539 *namelist-stmt* is NAMELIST / *namelist-group-name* / *namelist-group-object-list* ■
■ [[,] / *namelist-group-name* / *namelist-group-object-list*]...

R540 *namelist-group-object* is *variable-name*

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

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

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

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

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

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

An example of a NAMELIST statement is:

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

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

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

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

R541 *equivalence-stmt* **is** EQUIVALENCE *equivalence-set-list*

R542 *equivalence-set* **is** (*equivalence-object* , *equivalence-object-list*)

R543 *equivalence-object* **is** *variable-name*
or *array-element*
or *substring*

Constraint: An *equivalence-object* must not be a dummy argument, a pointer, an allocatable array, an automatic object, or a subobject of any such object, or a function name.

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

Constraint: If an *equivalence-object* is of derived type, all of the objects in the list must be of the same derived type; or all of the objects must be default numeric of default logical type or be of derived type with numeric storage association; or all of the objects must be default character type or be of derived type with character storage association. Otherwise, the results are processor dependent.

Constraint: If an equivalence object is an object of intrinsic type with nondefault kind parameters, all of the objects must be of the same type or the result is processor dependent.

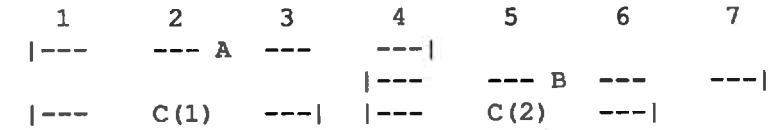
5.5.1.1 Equivalence Association. An EQUIVALENCE statement specifies that the storage sequences (14.6.2.1) of the data objects whose names appear in an *equivalence-set* are storage associated. All of the nonzero-sized sequences in the *equivalence-set*, if any, have the same first storage unit, and all of the zero-sized sequences in the *equivalence-set*, if any, are storage associated with one another and with the first storage unit of any nonzero-sized sequences. This causes the storage association of the data objects in the *equivalence-set* and may cause storage association of other data objects.

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

An EQUIVALENCE statement specifies that the storage sequences of all the character data objects whose names appear in an *equivalence-set* are storage associated. All of the nonzero-sized sequences in the *equivalence-set*, if any, have the same first storage unit, and all of the zero-sized sequences in the *equivalence-set*, if any, are storage associated with one another and with the first storage unit of any nonzero-sized sequences. This causes the storage association of the data objects in the *equivalence-set* and may cause storage association of other data objects. Any adjacent characters in the associated data objects may have the same character storage unit and thus may be storage associated. In the example:

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

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



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

For a nonzero-sized array, the use of the array name unqualified by a subscript list in an EQUIVALENCE statement has the same effect as using an array element designator that identifies the first element of the array.

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

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

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

```
REAL A (2)
DOUBLE PRECISION D (2)
EQUIVALENCE (A (1), D (1)), (A (2), D (2)) ! NOT STANDARD CONFORMING
```

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

```
SUBROUTINE A
   USE MODULE, ONLY : XX
   INTEGER BB, EE
   EQUIVALENCE (EE, XX)  ! Not standard conforming
   ...
   CONTAINS
      SUBROUTINE C
      REAL DD
      EQUIVALENCE (BB, DD)  ! Not standard conforming
      END SUBROUTINE C
END SUBROUTINE A
```

5.5.2 COMMON Statement. The **COMMON statement** specifies blocks of physical storage, called **common blocks**, that may be accessed by any of the scoping units in an executable program. Thus, the COMMON statement provides a global data facility based on storage association (14.6.2). The common blocks specified by the COMMON statement may be named and are called **named common blocks**, or may be unnamed and are called **blank common**.

R544 *common-stmt* **is** COMMON [/ [*common-block-name*] /]
■ *common-block-object-list* ■
■ [[,] / [*common-block-name*] / ■
■ *common-block-object-list*]...

R545 *common-block-object* **is** *variable-name* [(*explicit-shape-spec-list*)]

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

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

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

Constraint: If an object of derived type appears in a common block, it must be a sequenced structure or a storage-associated structure.

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

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

The form *variable-name* (*explicit-shape-spec-list*) declares *variable-name* to have the DIMENSION attribute and spcifies the array properties that apply. If numeric or character storage associated objects appear in common, it is as if the individual components were enumerated directly in the common list.

Examples of COMMON statements are:

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

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

(1) A storage sequence is formed consisting of the sequence of storage sequences (14.6.3.1) of all data objects in the common block object lists for the common block. The order of the storage sequences is the same as the order of the appearance of the common block object lists in the scoping unit.

(2) The storage sequence formed in (1) is extended to include all storage units of any storage sequence associated with it by equivalence association. The sequence may be extended only by adding storage units beyond the last storage unit. Data objects associated with an entity in a common block are considered to be in that common block.

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

5.5.2.3 Common Association. Within an executable program, the common block storage sequences of all nonzero-sized common blocks with the same name have the same first storage unit and the common block storage sequences of all zero-sized common blocks with the same name are storage associated with one another. Within an executable program, the common block storage sequences of all nonzero-sized blank common blocks have the same first storage unit and the storage sequences of all zero-sized blank common blocks are associated with one another and with the first storage unit of any nonzero-sized blank common blocks. This results in the association of objects in different scoping units.

An object of type character must become associated only with objects of type character that have the same kind parameter. A sequenced object that is not storage associated must become associated only with an object of the same type. If a storage-associated object becomes associated with any entity of a different type: i) the entity is an object of derived type with numeric storage association, all or the other entites in the common block must be of default numeric type, be of default logical type, or be of derived type with numeric storage association; if the entity is an object of derived type with character storage association, all of the other entities must either be of default character type or of derived type with character storage association. Otherwise, the results are processor dependent.

An object with nondefault kind type parameters must become associated only with an object with the same nondefault kind type parameters or the result is processor dependent.

5.5.2.4 Differences between Named Common and Blank Common. A blank common block has the same properties as a named common block, except for the following:

(1) Execution of a RETURN or END statement may cause data objects in named common blocks to become undefined unless the common block name has been declared in a SAVE statement, but never causes data objects in blank common to become undefined (14.7.6).

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

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

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

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

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

6. USE OF DATA OBJECTS

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

A **variable** is a data object that may be defined and redefined during execution of an executable program.

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

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

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

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

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

R604 *char-variable* **is** *variable*

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

R605 *int-variable* **is** *variable*

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

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

A literal constant is a scalar denoted by a syntactic form which indicates its type, type parameters, and value. A named constant is a constant that has been associated with a name with the PARAMETER attribute (5.1.2.1, 5.2.7). A reference to a constant is always permitted; redefinition of a constant is never permitted.

For example, given the declarations:

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

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

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

A scalar has rank zero.

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

R606 *substring* **is** *parent-string* (*substring-range*)

R607 *parent-string* **is** *scalar-variable-name*
or *array-element*
or *scalar-structure-component*
or *scalar-constant*

R608 *substring-range* **is** [*scalar-int-expr*] : [*scalar-int-expr*]

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

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

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

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

Examples of character substrings are:

```
B (1) (1:5)          array element as parent string
P % NAME (1:1)       structure component as parent string
ID (4:9)             scalar variable name as parent string
'0123456789' (N:N)   character constant as parent string
```

6.1.2 Structure Components. A *structure-component* is one of the components of a structure (4.4).

R609 *structure-component* **is** *parent-structure* **%** *component-name*

R610 *parent-structure* **is** *scalar-variable-name*
or *array-variable-name*
or *array-element*
or *array-section*
or *structure-component*
or *named-constant*

Constraint: If *parent-structure* is an array, the component must not be an array.

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

Constraint: *component-name* must be a component from the derived-type definition of the type of *parent-structure*.

The type of the structure component is the same as the type declared for the component in the derived-type definition. Each type parameter, if any, of a structure component is declared for the component in the derived-type definition (4.4.1) and is a constant.

A structure component has the INTENT, TARGET, or VALUE attribute if the parent structure has the attribute. A structure component is a pointer only if the component was defined to have the pointer attribute.

The resulting data subobject is an array if either the parent structure is an array or the component is an array.

Examples of structure components are:

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

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

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

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

6.2.1.1 Array Constants and Variables. A whole array is either a named constant or variable. A **whole array named constant** is the name of a constant expression (5.1.2.1 and 5.2.7) that is an array. A **whole array variable** is the name of a variable that is an array; the name does not have a subscript list appended to it.

The appearance of a whole array variable in an executable construct specifies all the elements of the array (2.4.7). An assumed-size array is permitted to appear as a whole array in an executable construct only as an actual argument in certain procedure references. which?

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

6.2.2 Array Elements and Array Sections.

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

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

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

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

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

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

R613 *parent-array* **is** *array-name*
or *structure-component*

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

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

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

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

} doesn't allow triangular sections.

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

R618 *vector-subscript* **is** *int-expr*

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

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

```
REAL A (10, 10)
CHARACTER (LEN = 10) B (5, 5, 5)
```

A (1, 2) is an array element, A (1:N:2, M) is a rank-one array section, and B (:, :, :) (2:3) is an array of shape (5, 5, 5) whose elements are substrings of length 2 of the corresponding elements of B.

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

6.2.2.2 Array Element Order. The elements of an array form a sequence known as the **array element order**. The position of an array element in this sequence is determined by the subscript order value of the subscript list designating the element. The subscript order value is computed

from the formulas in Table 6.1.

Table 6.1 Subscript Order Value.

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

Notes for Table 6.1:

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

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

An

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

Each *subscript-triplet* and *vector-subscript* in the section subscript list indicates a sequence of subscripts which may be empty (6.2.2). Each subscript in such a sequence must be within the bounds for its dimension unless the sequence is empty. The array section is the set of elements from the array determined by all possible subscript lists obtainable from the single subscripts or sequences of subscripts specified by each section subscript.

The rank of the array section is the number of subscript triplets and vector subscripts in the section subscript list. The shape is the rank-one array whose ith element is the number of integer values in the sequence indicated by the ith subscript triplet or vector subscript. If any of these sequences is empty, the array section has size zero. The subscript order of the elements of an array section is that of the array data object that the array section represents.

6.2.2.4 Subscript Triplet. A subscript triplet designates a regular sequence of subscripts consisting of zero or more subscript values. The third expression in the subscript triplet is the increment between the subscript values and is called the **stride**. The subscripts and stride of a subscript triplet are optional. An omitted first subscript in a subscript triplet is equivalent to a subscript whose value is the lower bound for the array and an omitted second subscript is equivalent to the upper bound (6.2.2). An omitted stride is equivalent to a stride of one.

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

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

The stride must not be zero.

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

Note that a subscript in a subscript triplet need not be within the declared bound for that dimension if all values used in selecting the array elements are within the declared bound.

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

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

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

6.2.2.5 Vector Subscript. A *vector-subscript* designates a sequence of subscripts that are the values of the elements of the expression. Each element of the expression must be defined unless it has size zero. A **many-one array section** is an array section with a *vector-subscript* having two or more elements with the same value. A many-one array section must not appear on the left of the equals in an assignment statement.

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

U = (/ 1, 3, 2 /)
V = (/ 2, 1, 1, 3 /)

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

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

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

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

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

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

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

An internal file must not be an array section with a vector subscript. An array section with a vector subscript must not be argument associated with a dummy array that is defined or redefined.

6.3 Dynamic Association. Dynamic control over the creation, association, and deallocation of pointer targets is provided by the ALLOCATE, NULLIFY, and DEALLOCATE statements and pointer assignment. ALLOCATE (6.3.1) creates targets for pointers; pointer assignment (7.5.2) associates pointers with existing targets; NULLIFY (6.3.2) disassociates pointers from targets, and DEALLOCATE deallocates targets. Dynamic association applies to scalars and arrays of any type.

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

R619 *allocate-stmt* **is** ALLOCATE (*allocation-list* ■
■ [, STAT = *stat-variable*])

R620 *stat-variable* **is** *scalar-int-variable*

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

R621 *allocation* **is** *allocate-name* [(*explicit-shape-spec-list*)]

Constraint: *allocate-name* must be the name of a pointer or an allocatable array.

Constraint: A bound in an *allocation explicit-shape-spec* is not restricted to a specification expression, but must not be an expression involving as a primary an array inquiry function (13.10.15) whose argument is any other object in the same ALLOCATE statement.

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

An example of an ALLOCATE statement is:

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

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

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

If an error condition occurs during execution of an ALLOCATE statement that does not contain the STAT= specifier, execution of the executable program is terminated. The ALLOCATED function (13.13.9) provides a mechanism for determining if an allocatable object is currently allocated.

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

Following successful execution of an ALLOCATE statement for a pointer, the pointer is associated with the target and may be used to reference or define the target. Additional pointer names may become associated with the pointer target or a part of the pointer target by pointer assignment. It is not an error to allocate a pointer that is currently associated with a target. In this case, a new pointer target is created as required by the attributes of the pointer and any array bounds specified in the ALLOCATE statement. The pointer is then associated with this new target. Any previous association with a target is broken. If the previous target had been created by allocation, it becomes inaccessible unless it can still be referred to by other pointer names that are currently associated with it.

At the beginning of execution of a function whose result is a pointer, the result pointer is disassociated. Before such a function returns, it must associate a target with this pointer.

6.3.2 NULLIFY Statement. The **NULLIFY statement** causes a pointer to be disassociated.

R622 *nullify-stmt* **is** NULLIFY (*pointer-name-list*)

Constraint: Each *pointer-name* must be the name of a pointer.

6.3.3 DEALLOCATE Statement. The **DEALLOCATE statement** causes the target of a pointer or an allocatable array to be deallocated and the pointer to be disassociated.

R623 *deallocate-stmt* **is** DEALLOCATE (*allocate-name-list* ■
■ [, STAT = *stat-variable*])

Constraint: Each *allocate-name* must be the name of a pointer or an allocatable array.

Constraint: If the *stat-variable* is a pointer, it must not be deallocated within the same DEALLOCATE statement.

Deallocating a pointer that is diassociated or whose target was not created by an ALLOCATE statement causes an error condition in the DEALLOCATE statement.

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

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

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

Host?

(1) An allocatable array with the SAVE attribute, or

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

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

When the execution of a procedure is terminated by execution of a RETURN or END statement, a pointer declared in the procedure retains its association with its target only if it is one of the following:

Host?

(1) A pointer dummy argument,

(2) A pointer with the SAVE attribute, or

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

An example of a DEALLOCATE statement is:

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

6.3.4 Summary of Array Name Appearances.

Table 6.2 Allowed Appearances of Array Names.

Place of Appearance	Explicit Shape Array	Structure Component Array	Target Array	Allocatable Array	Assumed Shape Array	Assumed Size Array
dummy-arg	Yes	No	Yes	Yes	Yes	Yes
use-stmt	Yes	No	Yes	Yes	No	No
type-declaration-stmt	Yes	Yes	Yes	Yes	Yes	Yes
namelist-stmt	Yes	No	No	No	No	No
equivalence-stmt	Yes	No	No	No	No	No
data-stmt	Yes	No	No	No	No	No
common-stmt	Yes	No	No	No	No	No
input-item-list or *output-item-list*	Yes	Yes	Yes	Yes	Yes	No
internal-file-unit	Yes	Yes	Yes	Yes	Yes	No
format	Yes	Yes	Yes	Yes	Yes	Yes
save-stmt	Yes	No	Yes	Yes	No	No
primary	Yes	Yes	Yes	Yes	Yes	No
assignment-stmt	Yes	Yes	Yes	Yes	Yes	No
pointer-assignment-stmt	No	No	Yes	No	No	No
allocate-stmt	No	No	Yes	Yes	No	No
deallocate-stmt, *free-stmt*	No	No	Yes	Yes	No	No
actual-arg in a reference to a procedure	Yes	Yes	Yes	Yes	Yes	Yes

Function result? ?

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

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

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

Simple examples of a level-3 expression are:

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

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

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

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

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

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

Simple examples of a level-4 expression are:

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

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

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

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

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

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

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

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

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

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

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

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

or .NEQV.

Simple examples of a level-5 expression are:

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

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

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

7.1.1.7 General Form of an Expression. Expressions are level-5 expressions optionally involving defined binary operators.

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

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

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

Simple examples of an expression are:

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

More complicated examples of an expression are:

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

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

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

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

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

A **character intrinsic operation**, **relational intrinsic operation**, and **logical intrinsic operation** are similarly defined in terms of a *character intrinsic operator* (//), *relational intrinsic operator*

(.EQ., .NE., .GT., .GE., .LT., .LE., ==, <>, >, >=, <, and <=), and *logical intrinsic operator* (.AND., .OR., .NOT., .EQV., and .NEQV.), respectively.

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

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

(The symbols I, R, Z, C, and L stand for the types integer, real, complex, character, and logical, respectively. Where more than one type for x_2 is given, the type of the result of the operation is given in the same relative position in the next column.)

Intrinsic Operator *op*	Type of x_1	Type of x_2	Type of $[x_1]$ *op* x_2
unary +, −		I, R, Z	I, R, Z
binary +, −, *, /, **	I	I, R, Z	I, R, Z
	R	I, R, Z	R, R, Z
	Z	I, R, Z	Z, Z, Z
//	C	C	C
.EQ., .NE., ==, <>	I	I, R, Z	L, L, L
	R	I, R, Z	L, L, L
	Z	I, R, Z	L, L, L
	C	C	L
.GT., .GE., .LT., .LE. >, >=, <, <=	I	I, R	L, L
	R	I, R	L, L
	C	C	L
.NOT.		L	L
.AND., .OR., .EQV., NEQV.	L	L	L

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

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

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

An **extension operation** is a defined operation in which the operator is of the form *defined-unary-op* or *defined-binary-op*. Such an operator is called an **extension operator**. Note that the operator used in an extension operation may be overloaded in that a generic interface for the operator may specify more than one function.

Can the intrinsic operators be overloaded for intrinsic types?

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

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

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

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

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

R727 *int-expr* **is** *expr*

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

R728 *numeric-expr* **is** *expr*

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

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

If a pointer appears as a primary in an intrinsic operation, the associated target object is referenced. The type, type parameters, and shape of the primary are those of the current target. If the pointer is not associated with a target, it may appear as a primary only in a reference to a procedure whose corresponding dummy argument is declared to be a pointer.

7.1.4.1 Data Type, Type Parameters, and Shape of a Primary. The data type, type parameters, and shape of a primary are determined according to whether the primary is a constant, variable, array constructor, structure constructor, function reference, or parenthesized expression. If a primary is a constant, its type, type parameters, and shape are those of the constant (4.3). If it is a structure constructor, it is scalar and its type is determined by the constructor name. If it is an array constructor, its type, type parameters, and shape are as described in 4.5. If it is a variable or function reference, its type, type parameters, and shape are those of the variable (5.1.1, 5.1.2) or the function reference (12.4.2), respectively. Note that in the case of a function reference, the function may be generic (13.10) or overloaded (12.5.5), in which case its type, type parameters, and rank are determined by the types, type parameters, and ranks of its actual arguments. If a primary is a parenthesized expression, its type, type parameters, and shape are those of the expression.

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

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

An expression of an intrinsic type has a kind type parameter. An expression of type character also has a length type parameter. For an expression x_1 // x_2 where x_1 and x_2 are of type character, the length type parameter is the sum of the lengths of the operands and the kind type parameter is the kind type parameter of x_1, which must be the same as the kind type parameter of x_2. For an expression *op* x_2 where *op* is a numeric intrinsic unary operator and x_2 is of type integer, real, or complex, the kind type parameter of the expression is that of the operand. For an expression x_1 *op* x_2 where *op* is a numeric intrinsic binary operator with one operand of type integer and the other of type real or complex, the kind type parameter of the expression is that of the real or complex operand. In the case where both operands are integer with kind type parameters k_1 and k_2, the kind type parameter of the expression is max (k_1, k_2). In the case where both operands are any of type real or complex with kind type parameters k_1 and k_2, the kind type parameter of the expression is max (k_1, k_2). In the case where both operands are of type logical with kind type parameters k_1 and k_2, the kind type parameter of the expression is max (k_1, k_2).

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

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

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

The following is an example of a scalar expression:

```
Q + 2.3 * R
```

where Q and R are scalars.

The following is an example of an array expression:

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

where A and B are arrays.

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

(1) A constant,

(2) An array constructor where each element and the bounds and strides of each implied-do are expressions whose primaries are either constant expressions or implied-do variables,

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

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

(5) An inquiry function (13.10.1, 13.10.5, 13.10.6, 13.10.8, 13.10.9, 13.10.15) reference where each argument is either a constant expression or a variable whose type parameters or bounds inquired about are not assumed or defined by an ALLOCATE statement or a pointer assignment,

(6) A subobject of a constant where each subscript, section subscript, substring starting point, and substring ending point is a constant expression, or

(7) A constant expression enclosed in parentheses.

A **restricted constant expression** is an expression in which each operator is an intrinsic operator. The exponentiation operator is permiitted only with an integer power. Each primary in a restricted constant expression must be one of the following:

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

(2) An array constructor where each element and the bounds and strides of each implied-do are expressions whose primaries are either restricted constant expressions or implied-do variables,

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

(4) An intrinsic function reference of type integer or character, except for functions in the classes Vector and Matrix Multiply Functions (13.10.13), Array Reduction Functions (13.10.14), Array Construction Functions (13.10.16), Array Manipulation Functions (13.10.18), and Array Location Functions (13.10.19). In the intrinsic function reference, each argument must be a restricted constant expression of type integer or character.

(5) A RESHAPE function reference where each argument is a restricted constant expression,

(6) An inquiry function reference where each argument is either a restricted constant expression or a variable whose type parameters or bounds inquired about are restricted constant expressions. Note that this excludes the PRESENT function because the object of the inquiry is neither the type parameter nor the bounds, or

(7) A restricted constant expression enclosed in parentheses.

R729 *constant-expr* **is** *expr*

R730 *char-constant-expr* **is** *char-expr*

R731 *int-constant-expr* **is** *int-expr*

R732 *logical-constant-expr* **is** *logical-expr*

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

The following are examples of constant expressions:

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

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

7.1.6.2 Specification Expression. A **restricted expression** is an expression in which each operation is intrinsic and each primary is:

(1) A constant,

(2) A variable that is a dummy argument,

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

(4) A variable that is made accessible by use association or host association,

(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,

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

(7) An intrinsic function reference where each argument is either a restricted expression or a variable whose type parameters or bounds inquired about are not assumed or defined by an ALLOCATE statement or a pointer assignment, or

(8) A restricted expression enclosed in parentheses.

R733 *specification-expr* **is** *scalar-int-expr*

A **specification expression** (R509) is a restricted expression that is scalar and of type integer.

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

The following are examples of specification expressions:

```
LBOUND (B, 1) + 5
M + LEN (C)
2 * PRECISION (A)
```

where B, M, and C are dummy arguments, B is an assumed-shape array, and A is a real variable made accessible by a USE statement.

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

Any variable or function reference used as an operand in an expression must be defined at the time the reference is executed. If the variable is a pointer, it must be associated with a target object that is defined. An integer operand must be defined with an integer value rather than a statement label value. All of the characters in a character data object reference must be defined.

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

Any numeric operation whose result is not mathematically defined is prohibited in the execution of an executable program. Examples are dividing by zero and raising a zero-valued primary to a zero-valued or negative-valued power. Raising a negative-valued primary of type real to a real power is also prohibited.

The execution of a function reference must not alter the value of any other variable within the statement in which the function reference appears. The execution of a function reference in a statement must not define or redefine (14.7) the value of any variable in common (5.5.2) or any variable made accessible by use association or host association (11.3.2, 11.2.2) if the definition or redefinition affects the value of any other reference in the statement. However, execution of a function reference in the logical expression of an IF statement (8.1.2.4) or WHERE statement (7.5.3.1) is permitted to define variables in the statement that is executed when the value of the expression is true. For example, in the statements:

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

F or G may define X. If a function reference causes definition or undefinition of an actual argument of the function, that argument or any associated entities must not appear elsewhere in the same statement. For example, the statements

```
A (I) = F (I)
Y = G (X) + X
```

are prohibited if the reference to F defines or undefines I or the reference to G defines or undefines X.

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

Execution of an array element reference requires the evaluation of its subscripts. The type of an expression in which the array element reference appears does not affect, and is not affected by, the evaluation of its subscripts. Execution of an array section reference requires the evaluation of its section subscripts. The type of an expression in which an array section appears does not affect, and is not affected by, the evaluation of the array section subscripts. Execution of a substring reference requires the evaluation of its substring expressions. The type of an expression in which a substring appears does not affect, and is not affected by, the evaluation of the substring expressions. ~~Note that~~ it is not necessary for a processor to evaluate any subscript expressions or substring expressions for an array of zero size or a character entity of zero length.

The appearance of an array constructor requires the evaluation of the bounds and stride of any array constructor implied-dos it may contain. The type of an expression in which an array constructor appears does not affect, and is not affected by, evaluation of such bounds and stride

expressions.

When an intrinsic binary operation is applied to a scalar and an array or two arrays of the same shape, the operation is performed element-by-element on corresponding array elements of the array operands. For example, the array expression

```
A + B
```

produces an array the same shape as A and B. The individual array elements of the result have the values of the first element of A added to the first element of B, the second element of A added to the second element of B, etc. The processor may perform the element-by-element operations in any order.

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

7.1.7.1 Evaluation of Operands. It is not necessary for a processor to evaluate all of the operands of an expression if the value of the expression can be determined otherwise. This principle is most often applicable to logical expressions and zero-sized arrays, but it applies to all expressions. For example, in evaluating the expression

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

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

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

where X is of size zero and W is a function, the function reference W(Z) need not be evaluated. If a statement contains a function reference in a part of an expression that need not be evaluated, all entities that would have become defined in the execution of that reference become undefined at the completion of evaluation of the expression containing the function reference. In the preceding examples, evaluation of these expressions causes Z to become undefined if L or W defines its argument.

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

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

Two expressions of a numeric type are mathematically equivalent if, for all possible values of their primaries, their mathematical values are equal. However, mathematically equivalent expressions of numeric type may produce different computational results. For example, any difference between the values of the expressions (1./3.)*3. and 1. is a computational difference, not a mathematical difference.

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

The following are examples of expressions with allowable alternative forms that may be used by the processor in the evaluation of those expressions. A, B, and C represent arbitrary real or complex operands; I and J represent arbitrary integer operands; and X, Y, and Z represent arbitrary operands of numeric type.

Expression	Allowable Alternative Form

```
X + Y                 Y + X
X * Y                 Y * X
-X + Y                Y - X
X + Y + Z             X + (Y + Z)
X - Y + Z             X - (Y - Z)
X * A / Z             X * (A / Z)
X * Y - X * Z         X * (Y - Z)
A / B / C             A / (B * C)
A / 5.0                 0.2 * A
```

The following are examples of expressions with forbidden alternative forms that must not be used by a processor in the evaluation of those expressions.

Expression	Nonallowable Alternative Form
`I / 2`	`0.5 * I`
`X * I / J`	`X * (I / J)`
`(X + Y) + Z`	`(X + Y + Z)`
`I / J / A`	`I / (J * A)`
`(X + Y) + Z`	`X + (Y + Z)`
`(X * Y) - (X * Z)`	`X * (Y - Z)`
`X * (Y - Z)`	`X * Y - X * Z`

In addition to the parentheses required to establish the desired interpretation, parentheses may be included to restrict the alternative forms that may be used by the processor in the actual evaluation of the expression. This is useful for controlling the magnitude and accuracy of intermediate values developed during the evaluation of an expression. For example, in the expression

```
A + (B - C)
```

the parenthesized expression (B–C) must be evaluated and then added to A.

Note that the inclusion of parentheses may change the mathematical value of an expression. For example, the two expressions:

```
A * I / J
A * (I / J)
```

may have different mathematical values if I and J are of type integer.

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

```
Z + R + I
```

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

7.1.7.4 Evaluation of the Character Intrinsic Operation. The rules given in 7.2.2 specify the interpretation of a character intrinsic operation. A processor needs to evaluate only as much of the character intrinsic operation as is required by the context in which the expression appears. For example, the statements

```
CHARACTER (LEN = 2) C1, C2, C3, CF
C1 = C2 // CF (C3)
```

do not require the function CF to be evaluated, because only the value of C2 is needed to determine the value of C1.

7.1.7.5 Evaluation of Relational Intrinsic Operations. The rules given in 7.2.3 specify the interpretation of relational intrinsic operations. Once the interpretation of an expression has been established in accordance with those rules, the processor may evaluate any other expression that is relationally equivalent, provided that the integrity of parentheses in any numeric expression is not violated. For example, the processor may choose to evaluate the expression

```
I .GT. J
```

where I and J are integer variables, as

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

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

7.1.7.6 Evaluation of Logical Intrinsic Operations. The rules given in 7.2.4 specify the interpretation of logical intrinsic operations. Once the interpretation of an expression has been established in accordance with those rules, the processor may evaluate any other expression that is logically equivalent, provided that the integrity of parentheses is not violated. For example, for the variables L1, L2, and L3 of type logical, the processor may choose to evaluate the expression

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

as

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

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

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

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

7.2 Interpretation of Intrinsic Operations.

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

The type, type parameters, and interpretation of an expression that consists of an intrinsic unary or binary operation are independent of the context in which the expression appears. In particular, the type, type parameters, and interpretation of such an expression are independent of the type and type parameters of any other larger expression in which it appears. For example, if X is of type real, J is of type integer, and INT is the real-to-integer intrinsic conversion function, the expression INT (X + J) is an integer expression and X + J is a real expression.

7.2.1 Numeric Intrinsic Operations. A numeric operation is used to express a numeric computation. Evaluation of a numeric operation produces a numeric value. The permitted data types for operands of the numeric intrinsic operations are specified in 7.1.2.

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

Table 7.2 Interpretation of the Numeric Intrinsic Operators.

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

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

If x_1 and x_2 are of type integer and x_2 has a negative value, the interpretation of x_1 ** x_2 is the same as the interpretation of $1/(x_1$ ** ABS$(x_2))$, which is subject to the rules of integer division (7.2.1.1). For example, 2**(–3) has the value of 1/(2**3), which is zero.

7.2.1.1 Integer Division. One operand of type integer may be divided by another operand of type integer. Although the mathematical quotient of two integers is not necessarily an integer, Table 7.1 specifies that an expression involving the division operator with two operands of type integer is interpreted as an expression of type integer. The result of such an operation is the integer closest to the mathematical quotient and between zero and the mathematical quotient inclusively. For example, the expression (–8)/3 has the value (–2).

7.2.1.2 Complex Exponentiation. In the case of a complex value raised to a complex power, the value of the operation is the "principal value" determined by x_1 ** x_2 = EXP$(x_2$ * LOG$(x_1))$, where EXP and LOG are functions described in 13.13.

7.2.2 Character Intrinsic Operation. The character intrinsic operator // is used to concatenate two operands of type character with the same kind type parameter. Evaluation of the character intrinsic operation produces a result of type character.

The interpretation of the character intrinsic operator // when used to form an expression is given in Table 7.3, where x_1 denotes the operand to the left of the operator and x_2 denotes the operand to the right of the operator.

Table 7.3 Interpretation of the Character Intrinsic Operator //.

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

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

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

double negative

Evaluation of a relational intrinsic operation produces a result of type logical.

The interpretation of the relational intrinsic operators is given in Table 7.4, where x_1 denotes the operand to the left of the operator and x_2 denotes the operand to the right of the operator. The operators <, <=, >, >=, ==, and <> always have the same interpretations as the operators .LT., .LE., .GT., .GE., .EQ., and .NE., respectively.

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

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

The value of the numeric relational operation

x_1 *rel-op* x_2

is the value of the expression

$((x_1) - (x_2))$ *rel-op* 0 assumes $x_1 - x_2$ doesn't overflow

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

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

For a character relational intrinsic operation, the operands are compared one character at a time in order, beginning with the first character of each character operand. If the operands are of unequal length, the shorter operand is treated as if it were extended on the right with blanks to the length of the longer operand. If every character of x_1 is the same as the character in the corresponding position in x_2, x_1 is equal to x_2. Otherwise, at the first position where the character operands differ, the character operand x_1 is considered to be less than x_2 if the character value of x_1 at this position precedes the value of x_2 in the collating sequence (3.1.7); x_1 is greater than x_2 if the character value of x_1 at this position follows the value of x_2 in the collating sequence. Note that the collating sequence depends partially on the processor; however, the result of the use of the operators .EQ., .NE., ==, and <> does not depend on the collating sequence.

7.2.4 Logical Intrinsic Operations. A logical operation is used to express a logical computation. Evaluation of a logical operation produces a result of type logical. The permitted data types for operands of the logical intrinsic operations are specified in 7.1.2.

The logical operators and their interpretation when used to form an expression are given in Table 7.5, where x_1 denotes the operand to the left of the operator and x_2 denotes the operand to the right of the operator.

Table 7.5 Interpretation of the Logical Intrinsic Operators.

Operator	Representing	Use of Operator	Interpretation
.NOT.	Logical Negation	.NOT. x_2	True if x_2 is false
.AND.	Logical Conjunction	x_1 .AND. x_2	True if x_1 and x_2 are both true
.OR.	Logical Inclusive Disjunction	x_1 .OR. x_2	True if x_1 and/or x_2 is true
.NEQV.	Logical Non-equivalence	x_1 .NEQV. x_2	True if either x_1 or x_2 is true, but not both
.EQV.	Logical Equivalence	x_1 .EQV. x_2	True if both x_1 and x_2 are true or both are false

The values of the logical intrinsic operations are shown in Table 7.6.

Table 7.6 The Values of Operations Involving Logical Intrinsic Operators.

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

7.3 Interpretation of Defined Operations. The interpretation of a defined operation is provided by the function subprogram that defines the operation. The type, type parameters, and interpretation of an expression that consists of a defined operation are independent of the type and type parameters of any larger expression in which it appears.

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

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

(2) The interface to the function subprogram is explicit (12.3.2.1) with a *generic-spec* of OPERATOR (op),

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

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

(5) The rank of x_2, and its shape if it is an array, match those of d_2.

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

(1) The function subprogram is specified with a FUNCTION statement (12.5.2.2) that specifies two dummy arguments, d_1 and d_2,

(2) The interface to the function subprogram is explicit (12.3.2.1) with a *generic-spec* of OPERATOR (op),

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

(4) The type parameters, if any, of x_1 and x_2 match those of d_1 and d_2, respectively, and

(5) The ranks of x_1 and x_2, and their shapes if either or both are arrays, match those of d_1 and d_2, respectively.

7.4 Precedence of Operators. There is a precedence among the intrinsic and extension operations implied by the general form in 7.1.1, which determines the order in which the operands are combined, unless the order is changed by the use of parentheses. This precedence order is summarized in Table 7.7.

Table 7.7 Categories of Operations and Relative Precedences.

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

The precedence of a defined operation is that of its operator, whether it is an overloaded intrinsic operator or an extension operator.

For example, in the expression

```
-A ** 2
```

the exponentiation operator (**) has precedence over the negation operator (–); therefore, the operands of the exponentiation operator are combined to form an expression that is used as the operand of the negation operator. The interpretation of the above expression is the same as the interpretation of the expression

```
- (A ** 2)
```

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

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

have the same interpretations as the expressions

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

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

```
A * .INVERSE. B
- .INVERSE. (B)
```

As another example, in the expression

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

the general form (7.1.1) implies a higher precedence for the .AND. operator than the .OR. operator; therefore, the interpretation of the above expression is the same as the interpretation of the expression

```
A .OR. (B .AND. C)
```

An expression may contain more than one category of operator. For example, the logical expression

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

where A, B, and C are of type real, and L is of type logical, contains a numeric operator, a relational operator, and a logical operator. This expression would be interpreted the same as the expression

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

Note that if:

(1) The operator ** is overloaded to operate on type logical,

(2) The operator .STARSTAR. is defined to duplicate the function of ** on type real,

(3) .MINUS. is defined to duplicate the unary operator –, and

(4) L1 and L2 are type logical and X and Y are type real,

then in precedence: L1 ** L2 is higher than X * Y; X * Y is higher than X .STARSTAR. Y; and .MINUS. X is higher than –X.

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

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

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

7.5.1.1 General Form.

R734 *assignment-stmt* **is** *variable = expr*

where *variable* is defined in R601 and *expr* is defined in R723.

Examples of an assignment statement are:

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

An assignment statement is either intrinsic or defined.

7.5.1.2 Intrinsic Assignment Statement. An **intrinsic assignment statement** is an assignment statement where the shapes of *variable* and *expr* conform and where:

(1) The types of *variable* and *expr* are intrinsic, as specified in Table 7.8 for assignment, or

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

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

An **array intrinsic assignment statement** is an intrinsic assignment statement for which *variable* is an array. The *variable* must not be a many-one array section (6.2.2.5).

Table 7.8 Type Conformance for the Assignment Statement *variable = expr*

Type of *variable*	Type of *expr*
integer	integer, real, complex
real	integer, real, complex
complex	integer, real, complex
character	character of the same kind type parameter as *variable*
logical	logical
derived type	same derived type as *variable*

7.5.1.3 Defined Assignment Statement. A **defined assignment statement** is an assignment statement that is not an intrinsic assignment statement, and is defined by a subroutine whose interface is explicit (7.5.1.6).

7.5.1.4 Intrinsic Assignment Conformance Rules. For an intrinsic assignment statement, *variable* and *expr* must conform in shape, and if *expr* is an array, *variable* must also be an array. The types of *variable* and *expr* must conform with the rules of Table 7.8.

If *variable* is a pointer, it must be associated with a definable target such that the type, type parameters, and shape of the target and *expr* conform.

For a numeric intrinsic assignment statement, *variable* and *expr* may have different numeric types or different type parameters, in which case the value of *expr* is converted to the type and type parameters of *variable* according to the rules of Table 7.9.

Table 7.9 Numeric Conversion and Assignment Statement *variable = expr.*

Type of *variable*	Value Assigned
integer	INT (*expr*, KIND = *scalar-int-restricted-constant-expr*)
real	REAL (*expr*, KIND = *scalar-int-restricted-constant-expr*)
complex	CMPLX (*expr*, KIND = *scalar-int-restricted-constant-expr*)

(The functions INT, REAL, and CMPLX are the generic functions defined in 13.13.)

For a character intrinsic assignment statement, *variable* and *expr* must have the same kind type parameter value, but may have different length type parameters in which case the conversion of *expr* to the length of *variable* is:

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

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

7.5.1.5 Interpretation of Intrinsic Assignments. Execution of an intrinsic assignment causes, in effect, the evaluation of the expression *expr* and all expressions within *variable* (7.1.7), the possible conversion of *expr* to the type and type parameters of *variable* (Table 7.9), and the definition of *variable* with the resulting value. The execution of the assignment must have the same effect as if the evaluation of all operations in *expr* and, if present, all operations in the subscripts or section subscripts of *variable* occurred before any portion of *variable* is defined by the assignment. The evaluation of expressions within *variable* must neither affect nor be affected by the evaluation of *expr*.

If *variable* is a pointer, the value of *expr* is assigned to the target of *variable*.

Both *variable* and *expr* may contain references to any portion of *variable*. For example, in the character intrinsic assignment statement:

```
STRING (2:5) = STRING (1:4)
```

the assignment of the first character of STRING to the second character does not affect the evaluation of STRING (1:4). That is, if the value of STRING prior to the assignment was 'ABCDEF', the value following the assignment is 'AABCDF'.

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

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

```
A = B
```

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

For example, the following program segment results in the values of the elements of array X being reversed:

```
REAL X (10)
...
X (1:10) = X (10:1:-1)
```

An intrinsic derived-type assignment is performed as if each component of *expr* were assigned to the corresponding component of *variable* using pointer assignment (7.5.2) for pointer components, and intrinsic or derived-type assignment for nonpointer components. The processor may perform the component-by-component assignment in any order or by any means that has the same effect.

For an example of a derived-type intrinsic assignment statement, if C and D are of the same derived type with components S, T, U, and V of type integer, logical, character, and another derived type, respectively, the intrinsic assignment.

```
C = D
```

assigns D % S to C % S using the numeric assignment statement, D % T to C % T using the logical intrinsic assignment statement, D % U to C % U using the character intrinsic assignment statement, and D % V to C % V using the derived-type intrinsic assignment statement.

When *variable* is a subobject, the assignment does not affect the definition status or value of other parts of the object. For example, if *variable* is an array section, the assignment does not affect the definition status or value of the elements of the parent array not specified by the array section.

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

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

(1) The subroutine subprogram is specified with a SUBROUTINE statement (12.5.2.3) that specifies two dummy arguments, d_1 and d_2,

(2) The interface to the subroutine subprogram is explicit, (12.3.2.1) with a *generic-spec* of ASSIGNMENT,

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

(4) The type parameters, if any, of x_1 and x_2 match those of d_1 and d_2, respectively, and

(5) The ranks of x_1 and x_2, and their shapes if either or both are arrays, match those of d_1 and d_2, respectively. may d_1 & d_2 be scalars?

7.5.2 Pointer Assignment Statement.

R735 *pointer-assignment-stmt* **is** *pointer-name* => *target*

R736 *target* **is** *variable*

Constraint: The *pointer-name* must have the POINTER attribute. The target object must have one of the attributes TARGET or POINTER or it must be a subobject of an object with one of these attributes.

Constraint: The *target* must be of the same type, type parameters, and rank as the pointer.

A pointer assignment statement associates a pointer with a target. If the target is itself a pointer, the pointer is associated with the same object as the target. If the target is a pointer that is not currently associated, the pointer also becomes disassociated.

Any previous association between the pointer and a target is broken.

In addition to pointer assignment, a pointer becomes associated with a target by allocation of the pointer.

A pointer must not be referenced or defined unless it is associated with a target that may be referenced or defined.

The following are examples of pointer assignment statements.

```
PNTR_TO_CELL => FIRST_CELL
SIMPLE_NAME => STRUCTURE % SUBSTRUCT % COMPONENT
ROW => MAT2D (N, :)
WINDOW => MAT2D (I-1:I+1, J-1:J+1)
ROW => MAT2D (K, 5:5+K)
EVERY_OTHER => VECTOR (1:N:2)
```

Pointer assignment for a pointer component of a structure also may take place by execution of an assignment statement for the structure (7.5.1.5). X=Y implies Y2P ⇒ X2P?!

7.5.3 Masked Array Assignment—WHERE. The masked array assignment is used to mask the evaluation of expressions and assignment of values in array assignment statements, according to the value of a logical array expression.

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

R737 *where-stmt* **is** WHERE (*mask-expr*) *assignment-stmt*

R738 *where-construct* **is** *where-construct-stmt*
[*assignment-stmt*]...
[*elsewhere-stmt*

[*assignment-stmt*]...]
end-where-stmt

R739 *where-construct-stmt* **is** WHERE (*mask-expr*)

R740 *mask-expr* **is** *logical-expr*

R741 *elsewhere-stmt* **is** ELSEWHERE [(*mask-expr*)]

R742 *end-where-stmt* **is** END WHERE

Constraint: In each *assignment-stmt*, the *mask-expr* and the variable being defined must be arrays of the same shape.

Examples of a masked array assignment are:

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

WHERE (PRESSURE <= 1.0)
   PRESSURE = PRESSURE + INC_PRESSURE
   TEMP = TEMP - 5.0
ELSEWHERE
   RAINING = .TRUE.
END WHERE
```

7.5.3.2 Interpretation of Masked Array Assignments. When the *assignment-stmt* in a *where-stmt* is executed, the *expr* of the *assignment-stmt* is evaluated for all the elements where *mask-expr* is true and the result is assigned to the corresponding elements of *variable*. When a *where-construct* is executed, the *mask-expr* is evaluated and the result saved by the processor. Each *assignment-stmt* in the where block is evaluated, in sequence, as if it were WHERE (*mask-expr*) *assignment-stmt* and each *assignment-stmt* in the ELSEWHERE block is evaluated, in sequence, as if it were WHERE (.NOT. *mask-expr*) *assignment-stmt*.

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

If an elemental function reference (12.4.3) occurs in the *expr* of an *assignment-stmt* and is not an actual argument of a nonelemental function reference, the function is evaluated only for the elements corresponding to true values in *mask-expr* (false values after ELSEWHERE).

In a masked array assignment, only a WHERE statement or a WHERE construct statement may be a branch target statement. The value of *mask-expr* governs the masking in the execution of the WHERE construct; subsequent changes to entities in *mask-expr* have no effect on the masking. Execution of an END WHERE has no effect.

8. EXECUTION CONTROL

Control constructs are used to control the execution sequence. These constructs are executable constructs containing blocks and executable statements that are used to alter the execution sequence.

8.1 Executable Constructs Containing Blocks. The following are executable constructs that contain blocks and may be used to control the execution sequence:

(1) IF Construct

(2) CASE Construct

(3) DO Construct

There is also a nonblock form of the DO construct.

A **block** is a sequence of executable constructs that is treated as an integral unit.

R801 *block* **is** [*execution-part-construct*]...

Executable constructs may be used to control which blocks of a program are executed or how many times a block is executed. Blocks are always bounded by statements that are particular to the construct in which they are embedded; however, in some forms of the DO construct, a sequence of executable constructs without a terminating boundary statement must obey all other rules governing blocks (8.1.1). Note that a block may be empty; execution of an empty block has no effect.

Any of these three constructs may be named. If a construct is named, the name must be the first lexical token of the first statement of the construct and the last lexical token of the construct. In fixed source form, the name preceding the construct must be placed after column 6.

A statement **belongs** to the innermost construct in which it appears unless it includes a specified construct name, in which case it belongs to the named construct.

An example of a construct containing a block is:

```
IF (A > 0.0) THEN
   B = SQRT (A)   ! THESE TWO STATEMENTS
   C = LOG (A)    ! FORM A BLOCK.
END IF
```

8.1.1 Rules Governing Blocks.

8.1.1.1 Executable Constructs in Blocks. If a block contains an executable construct, the executable construct must be contained entirely within the block.

8.1.1.2 Control Flow in Blocks. Transfer of control to the interior of a block from outside the block is prohibited. Transfers within a block and transfers from the interior of a block to outside the block may occur. For example, if a statement inside the block has a statement label, a GO TO statement using that label may appear in the same block. Subroutine and function references may appear in a block (12.4.2, 12.4.3, 12.4.4, 12.4.5).

8.1.1.3 Execution of a Block. Execution of a block begins with the execution of the first executable construct in the block. Unless there is a transfer of control out of the block, the execution of the block is completed when the last executable construct in the sequence is executed. The action that takes place at the terminal boundary depends on the particular construct and on the block within that construct. It is usually a transfer of control.

8.1.2 IF Construct. The **IF construct** selects for execution no more than one of its constituent blocks. The **IF statement** controls the execution of a single statement.

8.1.2.1 Form of the IF Construct.

R802 *if-construct* **is** *if-then-stmt*
block
[*else-if-stmt*
block]...
[*else-stmt*
block]
end-if-stmt

R803 *if-then-stmt* **is** [*if-construct-name* :] IF (*scalar-logical-expr*) THEN

R804 *else-if-stmt* **is** ELSE IF (*scalar-logical-expr*) THEN [*if-construct-name*]

R805 *else-stmt* **is** ELSE [*if-construct-name*]

R806 *end-if-stmt* **is** END IF [*if-construct-name*]

Constraint: If an *if-construct-name* is present, the same name must be specified on both the *if-then-stmt* and the corresponding *end-if-stmt*. If an *if-construct-name* appears on the *if-then-stmt*, it may also optionally appear on any *else-if-stmt* or *else-stmt* belonging to that *if-construct*.

8.1.2.2 Execution of an IF Construct. At most one of the blocks contained within the IF construct is executed. If there is an ELSE statement in the construct, exactly one of the blocks contained within the construct will be executed. The scalar logical expressions are evaluated in the order of their appearance in the construct until a true value is found or an ELSE statement or END IF statement is encountered. If a true value or an ELSE statement is found, the block immediately following is executed and this completes the execution of the construct. The scalar logical expressions in any remaining ELSE IF statements of the IF construct are not evaluated. If none of the evaluated expressions is true and there is no ELSE statement, the execution of the construct is completed without the execution of any blocks within the construct.

An ELSE IF statement or an ELSE statement must not be a branch target statement. It is permissible to branch to an END IF statement from within the IF construct, and also from outside the construct. Execution of an END IF statement has no effect.

8.1.2.3 Examples of IF Constructs.

```
IF (CVAR .EQ. 'RESET') THEN
   I = 0; J = 0; K = 0
END IF
PROOF_DONE: IF (PROP) THEN
   WRITE (3, '("QED")')
   STOP
ELSE
   PROP = NEXTPROP
END IF PROOF_DONE

IF (A .GT. 0) THEN
   B = C/A
   IF (B .GT. 0) THEN
      D = 1.0
   END IF
ELSE IF (C .GT. 0) THEN
   B = A/C
   D = -1.0
ELSE
   B = ABS (MAX (A, C))
   D = 0
END IF
```

8.1.2.4 IF Statement. The IF statement controls a single action statement (R215).

R807 *if-stmt* **is** IF (*scalar-logical-expr*) *action-stmt*

Constraint: The *action-stmt* in the *if-stmt* must not be an *if-stmt*. why not?

Execution of an IF statement causes evaluation of the scalar logical expression. If the value of the expression is true, the action statement is executed. If the value is false, the action statement is not executed and execution continues as though a CONTINUE statement (8.3) were executed.

The execution of a function reference in the scalar logical expression is permitted to affect entities in the action statement.

An example of an IF statement is:

```
IF (A > 0.0) A = LOG (A)
```

8.1.3 CASE Construct. The **CASE construct** selects for execution at most one of its constituent blocks.

8.1.3.1 Form of the CASE Construct. is WHERE redundant?

R808 *case-construct* **is** *select-case-stmt*
[*case-stmt*
block]...
end-select-stmt

R809 *select-case-stmt* **is** [*select-construct-name* :] SELECT CASE (*case-expr*)

R810 *case-stmt* **is** CASE *case-selector* [*select-construct-name*]

R811 *end-select-stmt* **is** END SELECT [*select-construct-name*]

Constraint: If a *select-construct-name* is present, the same name must be specified on both the *select-case-stmt* and the corresponding *end-select-stmt*. If a *select-construct-name* appears on the *select-case-stmt*, it may also optionally appear on any *case-stmt* belonging to that *case-construct*.

R812 *case-expr* **is** *scalar-int-expr* delete logical
or *scalar-char-expr* allow real
or *scalar-logical-expr*

R813 *case-selector* **is** (*case-value-range-list*)
or DEFAULT

Constraint: Only one DEFAULT *case-selector* may belong to any given *case-construct*.

R814 *case-value-range* **is** *case-value*
or *case-value* :
or : *case-value*
or *case-value* : *case-value*

R815 *case-value* **is** *scalar-int-constant-expr*
or *scalar-char-constant-expr*
or *scalar-logical-constant-expr*

Constraint: For a given *case-construct*, each *case-value* must be of the same type as *case-expr*. For character type, length differences are allowed, but the kind type parameters must be the same.

Constraint: A *case-value-range* using a colon must not be used if *case-expr* is of type logical.

Constraint: For a given *case-construct*, the *case-value-ranges* must not overlap; that is, there must be no possible value of the *case-expr* that matches more than one *case-value-range*.

8.1.3.2 Execution of a CASE Construct. The execution of the SELECT CASE statement causes the case expression to be evaluated. The resulting value is called the **case index** and must match no more than one of the selectors of one of the CASE statements of the construct. For a case value range list, a match occurs if the case index matches any of the case value ranges in the list. For a case index with a value of *c*, a match is determined as follows:

(1) If the case value range contains a single value *v* without a colon, a match occurs for data type logical if the expression *c* .EQV. *v* is true, and a match occurs for data type integer or character if the expression *c* .EQ. *v* is true.

(2) If the case value range is of the form *low* : *high*, a match occurs if the expression *low* .LE. *c* .AND. *c* .LE. *high* is true.

(3) If the case value range is of the form *low* :, a match occurs if the expression *low* .LE. *c* is true.

(4) If the case value range is of the form : *high*, a match occurs if the expression *c* .LE. *high* is true.

(5) If no other selector matches and a DEFAULT selector is present, it matches the case index.

(6) If no other selector matches and the DEFAULT selector is absent, there is no match. Execution continues with the statement following the CASE construct.

The block following the CASE statement containing the matching selector is executed. This completes execution of the construct.

At most one of the blocks of a CASE construct is executed.

A CASE statement must not be a branch target statement. It is permissible to branch to an END SELECT statement only from within the CASE construct.

8.1.3.3 Examples of CASE Constructs. An integer signum function:

```
INTEGER FUNCTION SIGNUM (N)
SELECT CASE (N)
CASE (:-1)
   SIGNUM = -1
CASE (0)
   SIGNUM = 0
CASE (1:)
   SIGNUM = 1
END SELECT
END
```

A code fragment to check for balanced parentheses:

```
CHARACTER (80) :: LINE
...
LEVEL=0
DO I = 1, 80
   CHECK_PARENS: SELECT CASE (LINE(I:I))
   CASE ('(')
      LEVEL = LEVEL + 1
   CASE (')')
      LEVEL = LEVEL - 1
      IF (LEVEL .LT. 0) THEN
         PRINT *, 'UNEXPECTED RIGHT PARENTHESIS'
         EXIT
      END IF
   CASE DEFAULT
      !IGNORE ALL OTHER CHARACTERS
```

```
      END SELECT CHECK_PARENS
   END DO
   IF (LEVEL .GT. 0) THEN
      PRINT *, 'MISSING RIGHT PARENTHESIS'
   END IF
```

The following three fragments are equivalent:

```
   IF (SILLY .EQ. 1) THEN
      CALL THIS
   ELSE
      CALL THAT
   END IF

   SELECT CASE (SILLY .EQ. 1)
   CASE (.TRUE.)
      CALL THIS
   CASE (.FALSE.)
      CALL THAT
   END SELECT

   SELECT CASE (SILLY)
   CASE DEFAULT
      CALL THAT
   CASE (1)
      CALL THIS
   END SELECT
```

A code fragment showing several selections of one block:

```
   SELECT CASE (N)
   CASE (1, 3:5, 8)  ! SELECTS 1, 3, 4, 5, 8
      CALL SUB
   CASE DEFAULT
      CALL OTHER
   END SELECT
```

8.1.4 DO Construct. The DO construct specifies the repeated execution of a sequence of executable constructs. Such a repeated sequence is called a **loop**. The EXIT and CYCLE statements may be used to modify the execution of a loop.

The number of iterations of a loop may be determined at the beginning of execution of the DO construct, or may be left indefinite ("DO forever"). In either case, an EXIT statement (8.1.4.4.4) may be executed anywhere within the DO construct to terminate the loop immediately. A particular iteration of the loop may be abbreviated by executing a CYCLE statement (8.1.4.4.3).

8.1.4.1 Forms of the DO Construct. The DO construct may be written in either a block form or a nonblock form.

R816 *do-construct* **is** *block-do-construct*
or *nonblock-do-construct*

8.1.4.1.1 Form of the Block DO Construct.

R817 *block-do-construct* **is** *do-stmt*
do-block
end-do

R818 *do-stmt* **is** *label-do-stmt*
or *nonlabel-do-stmt*

R819 *label-do-stmt* **is** [*do-construct-name* :] DO *label* [*loop-control*]

R820 *nonlabel-do-stmt* **is** [*do-construct-name* :] DO [*loop-control*]

R821 *loop-control* **is** [,] *do-variable* = *scalar-numeric-expr* , ■
■ *scalar-numeric-expr* [, *scalar-numeric-expr*]
or [,] WHILE (*scalar-logical-expr*)

R822 *do-variable* **is** *scalar-variable*

Constraint: The *do-variable* must be a scalar integer, default real, or double precision real named variable.

Constraint: Each *scalar-numeric-expr* in *loop-control* must be of type integer, default real, or double precision real.

R823 *do-block* **is** *block*

R824 *end-do* **is** *end-do-stmt*
or *continue-stmt*

R825 *end-do-stmt* **is** END DO [*do-construct-name*]

Constraint: If the *do-stmt* of a *block-do-construct* is identified by a *do-construct-name*, the corresponding *end-do* must be an *end-do-stmt* specifying the same *do-construct-name*. If the *do-stmt* of a *block-do-construct* does not so specify a *do-construct-name*, the corresponding *end-do* must not specify a *do-construct-name*.

Constraint: If the *do-stmt* is a *nonlabel-do-stmt*, the corresponding *end-do* must be an *end-do-stmt*.

Constraint: If the *do-stmt* is a *label-do-stmt*, the corresponding *end-do* must be identified with the same *label*.

8.1.4.1.2 Form of the Nonblock DO Construct.

R826 *nonblock-do-construct* **is** *action-term-do-construct*
or *outer-shared-do-construct*

R827 *action-term-do-construct* **is** *label-do-stmt*
do-body
do-term-action-stmt

R828 *do-body* **is** [*execution-part-construct*]...

R829 *do-term-action-stmt* **is** *action-stmt*

Constraint: A *do-term-action-stmt* must not be a *continue-stmt*, a *goto-stmt*, a *return-stmt*, a *stop-stmt*, an *exit-stmt*, a *cycle-stmt*, an *arithmetic-if-stmt*, or an *assigned-goto-stmt*.

Constraint: The *do-term-action-stmt* must be identified with a label and the corresponding *label-do-stmt* must refer to the same label.

R830 *outer-shared-do-construct* **is** *label-do-stmt*
do-body
shared-term-do-construct

R831 *shared-term-do-construct* **is** *outer-shared-do-construct*
or *inner-shared-do-construct*

R832 *inner-shared-do-construct* **is** *label-do-stmt*
do-body
do-term-shared-stmt

R833 *do-term-shared-stmt* **is** *action-stmt*

Constraint: A *do-term-shared-stmt* must not be a *goto-stmt*, a *return-stmt*, a *stop-stmt*, an *exit-stmt*, a *cycle-stmt*, an *arithmetic-if-stmt*, or an *assigned-goto-stmt*.

Constraint: The *do-term-shared-stmt* must be identified with a label and all of the *label-do-stmts* of the *shared-term-do-construct* must refer to the same label.

The *do-term-action-stmt*, *do-term-shared-stmt*, or *shared-term-do-construct* following the *do-body* of a nonblock DO construct is called the **DO termination** of that construct.

Within a scoping unit, all DO constructs whose DO statements refer to the same label are nonblock DO constructs, and are said to **share** the statement identified by that label.

8.1.4.2 Range of the DO Construct. The **range** of a block DO construct is the *do-block* which must satisfy the rules for blocks (8.1.1). In particular, transfer of control to the interior of such a block from outside the block is prohibited. It is permissible to branch to the END DO statement or CONTINUE statement terminating a block DO construct only from within the range of that DO construct.

The **range** of a nonblock DO construct consists of the *do-body* and the following DO termination. The end of such a range is not bounded by a particular statement as for the other executable constructs (e.g., END IF); nevertheless, the range satisfies the rules for blocks (8.1.1). Transfer of control into the *do-body* or to the DO termination from outside the range is prohibited; in particular, it is permissible to branch to a *do-term-shared-stmt* only from within the range of the corresponding *inner-shared-do-construct*.

8.1.4.3 Active and Inactive DO Constructs. A DO construct is either **active** or **inactive**. Initially inactive, a DO construct becomes active only when its DO statement is executed.

Once active, the DO construct becomes inactive only when the construct it specifies is terminated (8.1.4.4.4). When an active DO construct becomes inactive, the *do-variable*, if any, retains its last defined value.

8.1.4.4 Execution of a DO Construct. A DO construct specifies a loop, that is, a sequence of executable constructs that is executed repeatedly. There are three phases in the execution of a DO construct: initiation of the loop, execution of the loop range, and termination of the loop.

8.1.4.4.1 Loop Initiation. When the DO statement is executed, the DO construct becomes active. If there is a *loop-control* of the form

do-variable = *scalar-numeric-expr*$_1$, *scalar-numeric-expr*$_2$ [, *scalar-numeric-expr*$_3$]

the following steps are performed in sequence:

(1) The initial parameter m_1, the terminal parameter m_2, and the incrementation parameter m_3 are established by evaluating *scalar-numeric-expr*$_1$, *scalar-numeric-expr*$_2$, and *scalar-numeric-expr*$_3$, respectively, including, if necessary, conversion to the type of the *do-variable* according to the rules for numeric conversion (Table 7.9). If *scalar-numeric-expr*$_3$ does not appear, m_3 has a value of one. The value m_3 must not be zero.

(2) The DO variable becomes defined with the value of the initial parameter m_1.

(3) The **iteration count** is established and is the value of the expression

MAX (INT $((m_2 - m_1 + m_3) / m_3)$, 0)

Note that the iteration count is zero whenever:

$m_1 > m_2$ and $m_3 > 0$, or
$m_1 < m_2$ and $m_3 < 0$.

If *loop-control* is omitted, no iteration count is calculated. The effect is as if a large positive iteration count, impossible to decrement to zero, were established.

At the completion of the execution of the DO statement, the execution cycle begins.

8.1.4.4.2 The Execution Cycle. The **execution cycle** of a DO construct consists of the following steps performed in sequence:

(1) The iteration count is, if any, is tested. If the iteration count is zero, the loop terminates and the DO construct becomes inactive. If, as a result, all of the DO constructs sharing the *do-term-stmt* or ~~*continue-stmt*~~ are inactive, the execution of all of these constructs is complete. However, if some of the DO constructs sharing the *do-term-stmt* ~~or *continue-stmt*~~ are active, execution continues with step (3) of the execution cycle of the active DO construct whose

DO statement was most recently executed. Is this F77 compatible?

(2) If the iteration count is nonzero, the range of the loop is executed.

(3) The iteration count, if any, is decremented by one. The DO variable, if any, is incremented by the value of the incrementation parameter m_3.

(4) This cycle is executed repeatedly from step (1) until the loop is terminated.

Except for the incrementation of the DO variable that occurs in step (3), the DO variable must neither be redefined nor become undefined while the DO construct is active.

8.1.4.4.3 CYCLE Statement. Step (2) in the above execution cycle may be abbreviated by executing a CYCLE statement from within the range of the loop.

R834 *cycle-stmt* **is** CYCLE [*do-construct-name*]

Constraint: If a *cycle-stmt* refers to a *do-construct-name*, it must be within the range of that *do-construct*; otherwise, it must be within the range of at least one *do-construct*

A CYCLE statement belongs to a particular DO construct. If the CYCLE statement refers to a DO construct name, it belongs to that DO construct; otherwise, it belongs to the innermost DO construct in which it appears.

Execution of a CYCLE statement causes immediate progression to step (3) of the current execution cycle of the DO construct to which it belongs. If this construct is a nonblock DO construct, the *do-term-action-stmt* or *do-term-shared-stmt* is not executed.

In a block DO construct, a transfer of control to the *end-do* has the same effect as execution of a CYCLE statement belonging to that construct. In a nonblock DO construct, transfer of control to the *do-term-action-stmt* or *do-term-shared-stmt* causes that statement or construct itself to be executed. Unless a further transfer of control results, step (3) of the current execution cycle of the DO construct is then executed, with progression to step (4).

8.1.4.4.4 Loop Termination. The EXIT statement provides one way of terminating a ~~loop~~ construct.

R835 *exit-stmt* **is** EXIT [~~*do-construct-name*~~]

Constraint: If an *exit-stmt* refers to a ~~*do-construct-name*~~, it must be within the range of that ~~*do-construct*~~; otherwise, it must be within the range of at least one ~~*do-construct*~~.

An EXIT statement belongs to a particular ~~DO~~ construct. If the EXIT statement refers to a ~~DO~~ construct name, it belongs to that ~~DO~~ construct; otherwise, it belongs to the innermost ~~DO~~ construct in which it appears.

The ~~loop~~ construct terminates, and the DO construct becomes inactive, when any of the following occurs:

(1) Determination that the iteration count is zero when tested during step (1) of the above execution cycle.

(2) Execution of an EXIT statement belonging to the DO construct.

(3) Execution of an EXIT statement or a CYCLE statement that is within the range of the DO construct, but that belongs to an outer ~~DO~~ construct.

(4) Transfer of control to a statement that is ~~neither the *end-do* nor~~ not within the range of the DO construct.

(5) Execution of a RETURN statement within the range of the DO construct.

(6) Execution of a STOP statement anywhere in the program; or termination of the program for any other reason.

8.1.4.5 Examples of DO Constructs. The following are all valid examples of block DO constructs.

Example 1:

```
DO I = 1, M
   DO J = 1, N
      C (I, J) = SUM (A (I, J, :) * B (:, I, J))
   END DO
END DO
```

This program fragment computes a tensor product of two arrays.

Example 2:

```
READ (IUN, '(1X, G14.7)', IOSTAT = IOS) X
DO; IF (IOS .NE. 0) EXIT    ! A "DO WHILE" loop
   IF (X .GE. 0.) THEN
      CALL SUBA (X)
      CALL SUBB (X)
           ...
      CALL SUBZ (X)
   ENDIF
   READ (IUN, '(1X, G14.7)', IOSTAT = IOS) X
END DO
```

The above program fragment contains a DO construct that does not have an iteration count. The loop will continue to execute until an end-of-file or input/output error is encountered, at which point the EXIT statement terminates the loop. When a negative value of X is read, the program skips immediately to the next READ statement, bypassing most of the range of the loop.

Example 3:

```
DO   ! A "DO WHILE + 1/2" loop
   READ (IUN, '(1X, G14.7)', IOSTAT = IOS) X
   IF (IOS .NE. 0) EXIT
   IF (X < 0.) CYCLE
   CALL SUBA (X)
   CALL SUBB (X)
        ...
   CALL SUBZ (X)
END DO
```

This program fragment behaves exactly the same as the previous one. However, the READ statement has been moved to the interior of the range, so that only one READ statement is required. Also, a CYCLE statement has been used to avoid an extra level of IF nesting.

Example 4:

```
      SUM = 0.0
      READ (IUN) N
      OUTER: DO L = 1, N              ! A DO with a construct name
         READ (IUN) IQUAL, M, ARRAY (1:M)
         IF (IQUAL < IQUAL_MIN) CYCLE OUTER    ! Skip inner loop
         INNER: DO 40 I = 1, M      !A DO with a label and a name
            CALL CALCULATE (ARRAY (I), RESULT)
            IF (RESULT < 0.0) CYCLE
            SUM = SUM + RESULT
            IF (SUM > SUM_MAX) EXIT OUTER
   40    END DO INNER
      END DO OUTER
```

The outer loop has an iteration count of MAX (N, 0), and will execute that number of times unless SUM exceeds SUM_MAX, in which case the EXIT OUTER statement terminates both loops. The inner loop is skipped by the first CYCLE statement if the quality flag, IQUAL, is too low. If CALCULATE returns a negative RESULT, the second CYCLE statement prevents it from being summed. Note that both loops have construct names and the inner loop also has a label. A

construct name is required on the EXIT statement in order to terminate both loops, but is optional on the CYCLE statements because each belongs to its innermost loop.

Example 5:

```
      N = 0
      DO 50, I = 1, 10
         J = I
         DO K = 1, 5
            L = K
            N = N + 1  ! This statement executes 50 times
         END DO        ! Nonlabeled DO inside a labeled DO
   50 CONTINUE
```

After execution of the above program fragment, I = 11, J = 10, K = 6, L = 5, and N = 50.

Example 6:

```
      N = 0
      DO I = 1, 10
         J= I
         DO 60, K = 5, 1  ! This inner loop is never executed
            L = K
            N = N + 1
   60    CONTINUE          ! Labeled DO inside a nonlabeled DO
      END DO
```

After execution of the above program fragment, I = 11, J = 10, K = 5, N = 0, and L is not defined by these statements.

The following are all valid examples of nonblock DO constructs:

Example 7:

```
      DO 70
         READ (IUN, '(1X, G14.7)', IOSTAT = IOS) X
         IF (IOS .NE. 0) EXIT
         IF (X < 0.) GOTO 70
         CALL SUBA (X)
         CALL SUBB (X)
              ...
         CALL SUBY (X)
         CYCLE
   70    CALL SUBNEG(X)  ! SUBNEG called only when X < 0.
```

This is not a block DO construct because it ends with a statement other than END DO or CONTINUE. The loop will continue to execute until an end-of-file or input/output error occurs.

Example 8:

```
      SUM = 0.0
      READ (IUN) N
      DO 80, L = 1, N
         READ (IUN) IQUAL, M, ARRAY (1:M)
         IF (IQUAL < IQUAL_MIN) M = 0  ! Skip inner loop
         DO 80 I = 1, M
            CALL CALCULATE (ARRAY (I), RESULT)
            IF (RESULT < 0.) CYCLE
            SUM = SUM + RESULT
            IF (SUM > SUM_MAX) GOTO 81
   80    CONTINUE  ! This CONTINUE shared by both loops
   81 CONTINUE
```

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 may still be used to skip negative values of RESULT.

Example 9:

```
      N = 0
      DO 100 I = 1, 10
         J = 1
         DO 100 K = 1, 5
            L = K
  100         N = N + 1  ! This statement executes 50 times
```

In this example, the two loops share an assignment statement. After execution of this program fragment, I = 11, J = 10, K = 6, L = 5, and N = 50.

Example 10:

```
      N = 0
      DO 200 I = 1, 10
         J = I
         DO 200 K = 5, 1  ! This inner loop is never executed
            L = K
  200       N = N + 1
```

This example is very similar to the previous one, except that the inner loop is never executed. After execution of this program fragment, I = 11, J = 10, K = 5, N = 0, and L is not defined by these statements.

8.1.5 EXIT statement

Is this F77 compatible?

8.2 Branching. Branching is used to alter the normal execution sequence. A branch causes a transfer of control from one statement in a scoping unit to a labeled branch target statement in the same scoping unit. A **branch target statement** is an *action-stmt*, an *end-program-stmt*, an *end-function-stmt*, an *end-subroutine-stmt*, an *if-then-stmt*, an *end-if-stmt*, a *select-case-stmt*, ~~an *end-select-case-stmt*~~, a *do-stmt*, ~~an *end-do-stmt*~~, a *do-term-action-stmt*, a *do-term-shared-stmt*, or a *where-construct-stmt*.

~~32~~ It is permissible to branch to an END SELECT statement only from within its CASE construct.

It is permissible to branch to an END IF statement from within its IF construct, and also from outside the construct.

It is permissible to branch to ~~an *end-do-stmt*~~, a *do-term-action-stmt*, or a *do-term-shared-stmt* only from within its DO construct.

8.2.1 Statement Labels. A **statement label** provides a means of referring to an individual statement. Any statement not forming part of another statement may be identified with a label, but only branch target statements, FORMAT statements, and DO terminations may be referred to by the use of statement labels (3.2.5).

8.2.2 GO TO Statement.

R836 *goto-stmt* **is** GO TO *label*

Constraint: The *label* must be the statement label of a branch target statement that appears in the same scoping unit as the *goto-stmt*.

Execution of a GO TO statement causes a transfer of control so that the branch target statement identified by the label is executed next.

8.2.3 Computed GO TO Statement.

R837 *computed-goto-stmt* **is** GO TO (*label-list*) [,] *scalar-int-expr*

Constraint: Each *label* in *label-list* must be the statement label of a branch target statement that appears in the same scoping unit as the *computed-goto-stmt*.

The same statement label may appear more than once in a label list.

Execution of a computed GO TO statement causes evaluation of the scalar integer expression. If this value is *i* such that $1 \leq i \leq n$ where *n* is the number of labels in *label-list*, a transfer of control occurs so that the next statement executed is the one identified by the *i*th label in the list of labels. If *i* is less than 1 or greater than *n*, the execution sequence continues as though a CONTINUE statement were executed.

8.2.4 ASSIGN and Assigned GO TO Statement.

R838 *assign-stmt* **is** ASSIGN *label* TO *scalar-int-variable*

Constraint: The *label* must be the statement label of a branch target statement or *format-stmt* that appears in the same scoping unit as the *assign-stmt*.

R839 *assigned-goto-stmt* **is** GO TO *scalar-int-variable* [[,] (*label-list*)]

Constraint: Each *label* in *label-list* must be the statement label of a branch target statement that appears in the same scoping unit as the *assigned-goto-stmt*.

Execution of an ASSIGN statement causes a statement label to be assigned to an integer variable. While defined with a statement label value, the integer variable must not be referenced in any other context. An integer variable defined with a statement label value may be redefined with a statement label value or an integer value.

When an input/output statement containing the integer variable as a format specifier (9.4.1.1) is executed, the integer variable must be defined with the label of a FORMAT statement.

At the time of execution of an assigned GO TO statement, the integer variable must be defined with the value of a statement label of a branch target statement that appears in the same scoping unit. Note that the variable may be defined with a statement label value only by an ASSIGN statement in the same scoping unit as the assigned GO TO statement.

The execution of the assigned GO TO statement causes a transfer of control so that the branch target statement identified by the statement label currently assigned to the integer variable is executed next.

If the parenthesized list is present, the statement label assigned to the integer variable must be one of the statement labels in the list. A label may appear more than once in the label list of an assigned GOTO statement.

8.2.5 Arithmetic IF Statement.

R840 *arithmetic-if-stmt* **is** IF (*scalar-numeric-expr*) *label, label, label*

Constraint: Each *label* must be the label of a branch target statement that appears in the same scoping unit as the *arithmetic-if-stmt*.

Constraint: The *scalar-numeric-expr* must not be of type complex.

The same label may appear more than once in one arithmetic IF statement.

Execution of an arithmetic IF statement causes evaluation of the numeric expression followed by a transfer of control. The branch target statement identified by the first label, the second label, or the third label is executed next as the value of the numeric expression is less than zero, equal to zero, or greater than zero, respectively.

8.3 CONTINUE Statement.

Execution of a CONTINUE statement has no effect.

R841 *continue-stmt* **is** CONTINUE

8.4 STOP Statement.

R842 *stop-stmt* **is** STOP [*stop-code*]

R843 *stop-code* **is** *scalar-char-constant*
or *digit* [*digit* [*digit* [*digit* [*digit*]]]]

Execution of a STOP statement causes termination of execution of the executable program. At the time of termination, the stop code, if any, is accessible. Leading zero digits are not significant.

what does this mean?
not in glossary or index.

8.5 PAUSE Statement.

R844 *pause-stmt* **is** PAUSE [stop-code]

Execution of a PAUSE statement causes a suspension of execution of the executable program. Execution must be resumable. At the time of suspension of execution, the stop code, if any, is accessible. Resumption of execution is not under control of the program. If execution is resumed, the execution sequence continues as though a CONTINUE statement were executed. Leading zero digits in the stop code are not significant.

9. INPUT/OUTPUT STATEMENTS

Input statements provide the means of transferring data from external media to internal storage or from an internal file to internal storage. This process is called **reading**. **Output statements** provide the means of transferring data from internal storage to external media or from internal storage to an internal file. This process is called **writing**. Some input/output statements specify that editing of the data is to be performed.

In addition to the statements that transfer data, there are auxiliary input/output statements to manipulate the external medium, or to describe or inquire about the properties of the connection to the external medium.

The input/output statements are the OPEN, CLOSE, READ, WRITE, PRINT, BACKSPACE, ENDFILE, REWIND, and INQUIRE statements.

The READ statement is a **data transfer input statement**. The WRITE statement and the PRINT statements are **data transfer output statements**. The OPEN statement and the CLOSE statement are **file connection statements**. The INQUIRE statement is a **file inquiry statement**. The BACKSPACE, ENDFILE, and REWIND statements are **file positioning statements.**

9.1 Records. A **record** is a sequence of values or a sequence of characters. For example, a line on a terminal is usually considered to be a record. However, a record does not necessarily correspond to a physical entity. There are three kinds of records:

(1) Formatted

(2) Unformatted

(3) Endfile

9.1.1 Formatted Record. A **formatted record** consists of a sequence of characters that are capable of representation in the processor. The length of a formatted record is measured in characters and depends primarily on the number of characters put into the record when it is written. However, it may depend on the processor and the external medium. The length may be zero. Formatted records may be read or written only by formatted input/output statements.

Formatted records may be prepared by means other than Fortran; for example, by some manual input device.

9.1.2 Unformatted Record. An **unformatted record** consists of a sequence of values in a processor-dependent form and may contain data of any type or may contain no data. The length of an unformatted record is measured in processor-dependent units and depends on the output list (9.4.2) used when it is written, as well as on the processor and the external medium. The length may be zero. Unformatted records may be read or written only by unformatted input/output statements.

9.1.3 Endfile Record. An **endfile record** is written explicitly by the ENDFILE statement; the file must be connected for sequential access. An endfile record is written implicitly to a file connected for sequential access when the last data transfer or file positioning statement referring to the file is a data transfer output statement, and:

(1) A REWIND or BACKSPACE statement references the unit, or

(2) The unit (file) is closed, either explicitly by a CLOSE statement or implicitly by a program termination not caused by an error condition.

An endfile record may occur only as the last record of a file. An endfile record does not have a length property.

9.2 Files. A **file** is a sequence of records.

There are two kinds of files:

(1) External

(2) Internal

9.2.1 External Files. An **external file** is any file that exists in a medium external to the executable program.

The records of a file are either all formatted or all unformatted, except that the last record of a file may be an endfile record. At any given time, there is a processor-dependent **set of allowed access methods**, a processor-dependent **set of allowed forms**, a processor-dependent **set of allowed actions**, and a processor-dependent **set of allowed record lengths** for a file.

A file may have a name; a file that has a name is called a **named file**. The name of a named file is a character string. The set of allowable names for a file is processor dependent.

An external file that is connected to a unit has a **position** property (9.2.1.3).

9.2.1.1 File Existence. At any given time, there is a processor-dependent set of external files that are said to **exist** for an executable program. A file may be known to the processor, yet not exist for an executable program at a particular time. For example, there may be security reasons that prevent a file from existing for an executable program. A file may exist and contain no records; an example is a newly created file not yet written.

To **create a file** means to cause a file to exist that did not exist previously. To **delete a file** means to terminate the existence of the file.

All input/output statements may refer to files that exist. An INQUIRE, OPEN, CLOSE, WRITE, PRINT, REWIND, or ENDFILE statement may also refer to a file that does not exist.

9.2.1.2 File Access. There are two methods of accessing the records of an external file, sequential and direct. Some files may have more than one allowed access method; other files may be restricted to one access method. For example, a processor may allow only sequential access to a file on magnetic tape. Thus, the set of allowed access methods depends on the file and the processor.

The method of accessing the file is determined when the file is connected to a unit (9.3.2) or when the file is created if the file is preconnected (9.3.3).

9.2.1.2.1 Sequential Access. When connected for sequential access, an external file has the following properties:

(1) The order of the records is the order in which they were written if the direct access method is not a member of the set of allowed access methods for the file. If the direct access method is also a member of the set of allowed access methods for the file, the order of the records is the same as that specified for direct access. In this case, the first record accessed by sequential access is the record whose record number is 1 for direct access. The second record accessed by sequential access is the record whose record number is 2 for direct access, etc. A record that has not been written since the file was created must not be read.

(2) The records of the file are either all formatted or all unformatted, except that the last record of the file may be an endfile record. Unless the previous data transfer or file positioning statement referring to the file was a data transfer output statement, the last record, if any, of the file must be an endfile record.

(3) The records of the file must not be read or written by direct access input/output statements.

9.2.1.2.2 Direct Access. When connected for direct access, an external file has the following properties:

(1) Each record of the file is uniquely identified by a positive integer called the **record number**. The record number of a record is specified when the record is written. Once established, the record number of a record can never be changed. Note that a record may not be deleted; however, a record may be rewritten. The order of the records is the order of their record numbers.

(2) The records of the file are either all formatted or all unformatted. If the sequential access method is also a member of the set of allowed access methods for the file, its endfile record, if any, is not considered to be part of the file while it is connected for direct access. If the sequential access method is not a member of the set of allowed access methods for the file, the file must not contain an endfile record.

(3) Reading and writing records is accomplished only by direct access input/output statements.

(4) All records of the file have the same length.

(5) Records need not be read or written in the order of their record numbers. Any record may be written into the file while it is connected to a unit. For example, it is permissible to write record 3, even though records 1 and 2 have not been written. Any record may be read from the file while it is connected to a unit, provided that the record has been written since the file was created.

(6) The records of the file must not be read or written using list-directed (10.8) or namelist formatting (10.9).

9.2.1.3 File Position. Execution of certain input/output statements affects the position of an external file. Certain circumstances can cause the position of a file to become indeterminate.

The **initial point** of a file is the position just before the first record. The **terminal point** is the position just after the last record. If there are no records in the file, the initial point and the terminal point are the same position. including endfile?

If a file is positioned within a record, that record is the **current record**; otherwise, there is no current record.

Let n be the number of records in the file. If $1 < i \leq n$ and a file is positioned within the ith record or between the $(i-1)$th record and the ith record, the $(i-1)$th record is the **preceding record**. If $n \geq 1$ and the file is positioned at its terminal point, the preceding record is the nth and last record. If $n = 0$ or if a file is positioned at its initial point or within the first record, there is no preceding record.

If $1 \leq i < n$ and a file is positioned within the ith record or between the ith and $(i+1)$th record, the $(i+1)$th record is the **next record**. If $n \geq 1$ and the file is positioned at its initial point, the first record is the next record. If $n = 0$ or if a file is positioned at its terminal point or within the nth (last) record, there is no next record.

9.2.1.3.1 Advancing and Nonadvancing Input/Output. An **advancing input/output statement** always positions the file after the last record read or written, unless there is an error condition.

A **nonadvancing input/output statement** may position the file at a character position within the current record. Using nonadvancing input/output, it is possible to read or write a record of the file by a sequence of input/output statements each accessing a portion of the record.

9.2.1.3.2 File Position Prior to Data Transfer. The positioning of the file prior to data transfer depends on the method of access: sequential or direct.

For sequential access on input, if the previous statement accessing the file performed nonadvancing input, the file position is not changed. Otherwise, the file is positioned at the

beginning of the next record and this record becomes the current record. Input must not occur if there is no next record or if the last statement accessing the file performed nonadvancing output.

If the file contains an endfile record, the file must not be positioned after the endfile record prior to data transfer. However, a REWIND or BACKSPACE statement may be used to reposition the file.

For sequential access on output, if the previous statement accessing the file performed nonadvancing output, the file position is not changed. Otherwise, a new record is created as the next record of the file; this new record becomes the last and current record of the file and the file is positioned at the beginning of this record.

For direct access, the file is positioned at the beginning of the record specified by the record specifier. This record becomes the current record.

9.2.1.3.3 File Position After Data Transfer. If an end-of-file condition (9.4.3) exists as a result of reading an endfile record and no error condition (9.4.3) exists, the file is positioned after the endfile record.

If the data transfer was a nonadvancing READ or WRITE statement, the file position is not changed if no error or end-of-file condition exists. For other data transfer input/output, if no error condition or end-of-file condition exists, the file is positioned after the record just read or written and that record becomes the preceding record.

If an error condition exists, the position of the file is indeterminate.

9.2.2 Internal Files. Internal files provide a means of transferring and converting data from internal storage to internal storage.

9.2.2.1 Internal File Properties. An internal file has the following properties:

(1) The file is a character variable.

(2) A record of an internal file is a scalar character variable.

(3) If the file is a scalar character variable, it consists of a single record whose length is the same as the length of the scalar character variable. If the file is a character array, it is treated as a sequence of character array elements. Each array element, if any, is a record of the file. The ordering of the records of the file is the same as the ordering of the array elements in the array (6.2.2.2) or the array section (6.2.2.3). Every record of the file has the same length, which is the length of an array element in the array.

(4) A record of the internal file becomes defined by writing the record. If the number of characters written in a record is less than the length of the record, the remaining portion of the record is filled with blanks. The number of characters to be written must not exceed the length of the record.

(5) A record may be read only if the record is defined.

(6) A record of an internal file may become defined (or undefined) by means other than an output statement. For example, the character variable may become defined by a character assignment statement.

(7) An internal file is always positioned at the beginning of the first record prior to data transfer. This record becomes the current record.

(8) On input, blanks are treated as though the format had an initial BN edit descriptor (10.6.6).

9.2.2.2 Internal File Restrictions. An internal file has the following restrictions:

(1) Reading and writing records must be accomplished only by sequential access formatted input/output statements that do not specify namelist formatting.

(2) An internal file must not be specified in a file connection statement, a file positioning statement, or a file inquiry statement.

9.3 File Connection. A **unit**, specified by an *io-unit*, provides a means for referring to a file.

R901 *io-unit* **is** *external-file-unit*
or *
or *internal-file-unit*

R902 *external-file-unit* **is** *scalar-int-expr*

R903 *internal-file-unit* **is** *char-variable*

Constraint: The *char-variable* must not be an array section with a vector subscript.

A unit is either an external unit or an internal unit. An **external unit** is used to refer to an external file and is specified by an *external-file-unit* or an asterisk. An **internal unit** is used to refer to an internal file and is specified by an *internal-file-unit*.

A scalar integer expression that identifies an external file unit must be zero or positive.

The *io-unit* in a file positioning statement, a file connection statement, or a file inquiry statement must be an *external-file-unit*.

The external unit identified by the value of the *scalar-int-expr* is the same external unit in all program units of the executable program. In the example:

```
SUBROUTINE A
   READ (6) X
   .
   .
   .
SUBROUTINE B
   N = 6
   REWIND N
```

the value 6 used in both program units identifies the same external unit.

An asterisk identifies a particular processor-dependent external unit that is preconnected for formatted sequential access (9.4.4.2).

9.3.1 Unit Existence. At any given time, there is a processor-dependent set of external units that are said to exist for an executable program. All input/output statements may refer to units that exist. The INQUIRE statement and the CLOSE statement also may refer to units that do not exist.

9.3.2 Connection of a File to a Unit. An external unit has a property of being **connected** or not connected. If connected, it refers to a file. An external unit may become connected by preconnection or by the execution of an OPEN statement. The property of connection is symmetric; if a unit is connected to a file, the file is connected to the unit.

All input/output statements except an OPEN, a CLOSE, or an INQUIRE statement must refer to a unit that is connected to a file and thereby make use of or affect that file.

A file may be connected and not exist. An example is a preconnected new external file.

A unit must not be connected to more than one file at the same time, and a file must not be connected to more than one unit at the same time. However, means are provided to change the status of an external unit and to connect a unit to a different file.

After an external unit has been disconnected by the execution of a CLOSE statement, it may be connected again within the same executable program to the same file or to a different file. After an external file has been disconnected by the execution of a CLOSE statement, it may be connected again within the same executable program to the same unit or to a different unit. Note,

however, that the only means of referencing a file that has been disconnected is by the appearance of its name in an OPEN or INQUIRE statement. There may be no means of reconnecting an unnamed file once it is disconnected.

An internal unit is always connected to the internal file designated by the *char-variable* that identifies the unit. Note, however, that if the *char-variable* is a pointer that is disassociated, an allocatable array not currently allocated or a subobject of such an array, the internal file must not be referenced by an input/output statement.

9.3.3 Preconnection. **Preconnection** means that the unit is connected to a file at the beginning of execution of the executable program and therefore it may be specified in input/output statements without the prior execution of an OPEN statement.

On input, blanks are treated as if the file had been opened with a BLANK = NULL specified in an OPEN statement (9.3.4.6).

9.3.4 The OPEN Statement. The **OPEN statement** may be used to connect an existing file to a unit, create a file that is preconnected, create a file and connect it to a unit, or change certain specifiers of a connection between a file and a unit. The file must be an external file.

An external unit may be connected by an OPEN statement in any program unit of an executable program and, once connected, a reference to it may appear in any program unit of the executable program.

If a unit is connected to a file that exists, execution of an OPEN statement for that unit is permitted. If the FILE= specifier is not included in such an OPEN statement, the file to be connected to the unit is the same as the file to which the unit is already connected.

If the file to be connected to the unit does not exist but is the same as the file to which the unit is preconnected, the properties specified by an OPEN statement become a part of the connection.

If the file to be connected to the unit is not the same as the file to which the unit is connected, the effect is as if a CLOSE statement without a STATUS= specifier had been executed for the unit immediately prior to the execution of an OPEN statement.

If the file to be connected to the unit is the same as the file to which the unit is connected, only the BLANK=, DELIM=, PAD=, ERR=, and IOSTAT= specifiers may have a value different from the one currently in effect. Execution of such an OPEN statement causes any new value of the BLANK=, DELIM=, or PAD= specifiers to be in effect, but does not cause any change in any of the unspecified specifiers and the position of the file is unaffected. The ERR= and IOSTAT= specifiers from any previously executed OPEN statement have no effect on any currently executed OPEN statement.

If a file is already connected to a unit, execution of an OPEN statement on that file and a different unit is not permitted.

R904 *open-stmt* **is** OPEN (*connect-spec-list*)

R905 *connect-spec* **is** [UNIT=] *external-file-unit*
or IOSTAT= *scalar-int-variable*
or ERR= *label*
or FILE= *file-name-expr*
or STATUS= *scalar-char-expr*
or ACCESS= *scalar-char-expr*
or FORM= *scalar-char-expr*
or RECL= *scalar-int-expr*
or BLANK= *scalar-char-expr*
or POSITION= *scalar-char-expr*
or ACTION= *scalar-char-expr*
or DELIM= *scalar-char-expr*
or PAD= *scalar-char-expr*

R906 *file-name-expr* **is** *scalar-char-expr*

Constraint: If the optional characters UNIT= are omitted from the unit specifier, the unit specifier must be the first item in the *connect-spec-list*.

Constraint: ~~Each~~ specifier ~~must not~~ appear more than once in a given *open-stmt*; an *external-file-unit* must be specified.

Constraint: The *label* used in the ERR= specifier must be the statement label of a branch target statement that appears in the same scoping unit as the OPEN statement.

If the STATUS= specifier has the value OLD or NEW, the FILE= specifier must be present. If the STATUS= specifier has the value SCRATCH, the FILE= specifier must be absent.

Except for the FILE= specifier, a specifier that requires a *scalar-char-expr* may have a limited list of character values. These values are listed for each such specifier. Any trailing blanks are ignored. If a processor is capable of representing letters in both upper and lower case, the value specified is without regard to case. Some specifiers have a default value if the specifier is omitted.

The IOSTAT= specifier and ERR= specifier are described in 9.4.1.4 and 9.4.1.5, respectively.

An example of an OPEN statement is:

```
OPEN (10, FILE = 'employee.names', ACTION = 'READ', PAD = 'YES')
```

9.3.4.1 FILE= Specifier in the OPEN Statement. The value of the FILE= specifier is the name of the file to be connected to the specified unit. Any trailing blanks are ignored. The *file-name-expr* must be a name that is allowed by the processor. If this specifier is omitted and the unit is not connected to a file, it may become connected to a processor-dependent file. If a processor is capable of representing letters in both upper and lower case, the interpretation of case is processor dependent.

9.3.4.2 STATUS= Specifier in the OPEN Statement. The *scalar-char-expr* must evaluate to OLD, NEW, SCRATCH, REPLACE, or UNKNOWN. If OLD is specified, the file must exist. If NEW is specified, the file must not exist.

Successful execution of an OPEN statement with NEW specified creates the file and changes the status to OLD. If REPLACE is specified and the file does not already exist, the file is created and the status is changed to OLD. If REPLACE is specified and the file does exist, the file is deleted, a new file is created with the same name, and the status is changed to OLD. If SCRATCH is specified, the file is created and connected to the specified unit for use by the executable program but is deleted at the execution of a CLOSE statement referring to the same unit or at the termination of the executable program. Note that SCRATCH must not be specified with a named file. If UNKNOWN is specified, the status is processor dependent. If this specifier is omitted, the default value is UNKNOWN.

9.3.4.3 ACCESS= Specifier in the OPEN Statement. The *scalar-char-expr* must evaluate to SEQUENTIAL or DIRECT. The ACCESS= specifier specifies the access method for the connection of the file as being sequential or direct. If this specifier is omitted, the default value is SEQUENTIAL. For an existing file, the specified access method must be included in the set of allowed access methods for the file. For a new file, the processor creates the file with a set of allowed access methods that includes the specified method.

9.3.4.4 FORM= Specifier in the OPEN Statement. The *scalar-char-expr* must evaluate to FORMATTED or UNFORMATTED. The FORM= specifier determines whether the file is being connected for formatted or unformatted input/output. If this specifier is omitted, the default value is UNFORMATTED if the file is being connected for direct access, and the default value is FORMATTED if the file is being connected for sequential access. For an existing file, the specified form must be included in the set of allowed forms for the file. For a new file, the processor creates the file with a set of allowed forms that includes the specified form.

9.3.4.5 RECL= Specifier in the OPEN Statement. The value of the RECL= specifier must be positive. It specifies the length of each record in a file being connected for direct access, or specifies the maximum length of a record in a file being connected for sequential access. This specifier must be present when a file is being connected for direct access. If this specifier is omitted when a file is being connected for sequential access, the default value is processor dependent. If the file is being connected for formatted input/output, the length is the number of characters. If the file is being connected for unformatted input/output, the length is measured in processor-dependent units. For an existing file, the value of the RECL= specifier must be included in the set of allowed record lengths for the file. For a new file, the processor creates the file with a set of allowed record lengths that includes the specified value.

9.3.4.6 BLANK= Specifier in the OPEN Statement. The *scalar-char-expr* must evaluate to NULL or ZERO. The BLANK= specifier is permitted only for a file being connected for formatted input/output. If NULL is specified, all blank characters in numeric formatted input fields on the specified unit are ignored, except that a field of all blanks has a value of zero. If ZERO is specified, all blanks other than leading blanks are treated as zeros. If this specifier is omitted, the default value is NULL.

9.3.4.7 POSITION= Specifier in the OPEN Statement. The *scalar-char-expr* must evaluate to ASIS, REWIND, or APPEND. The connection must be for sequential access. A file that did not exist previously (a new file, either specified explicitly or by default) is positioned at its initial point. REWIND positions an existing file at its initial point. APPEND positions an existing file such that the endfile record is the next record, if it has one. If an existing file does not have an endfile record, APPEND positions the file at its terminal point. ASIS leaves the position unchanged if the file exists and already is connected. ASIS leaves the position unspecified if the file exists but is not connected. If this specifier is omitted, the default value is ASIS.

9.3.4.8 ACTION= Specifier in the OPEN Statement. The *scalar-char-expr* must evaluate to READ, WRITE, or READWRITE. READ specifies that the WRITE, PRINT and ENDFILE statements must not refer to this connection. WRITE specifies that READ statements must not refer to this connection. READWRITE permits any I/O statements to refer to this connection. If this specifier is omitted, the default value is READWRITE. For an existing file, the specified action must be included in the set of allowed actions for the file. For a new file, the processor creates the file with a set of allowed actions that includes the specified action.

9.3.4.9 DELIM= Specifier in the OPEN Statement. The *scalar-char-expr* must evaluate to APOSTROPHE, QUOTE, or NONE. If APOSTROPHE is specified, the apostrophe will be used to delimit character constants written with list-directed or namelist formatting and all internal apostrophes will be doubled. If QUOTE is specified, the quotation mark will be used to delimit character constants written with list-directed or namelist formatting and all internal quotation marks will be doubled. If the value of this specifier is NONE, a character constant when written will not be delimited by apostrophes or quotation marks, nor will any internal apostrophes or quotation marks be doubled. If this specifier is omitted, the default value is NONE. This specifier is permitted only for a file being connected for formatted input/output. This specifier is ignored during input of a formatted record.

9.3.4.10 PAD= Specifier in the OPEN Statement. The *scalar-char-expr* must evaluate to YES or NO. If YES is specified, a formatted input record is logically padded with blanks when an input list is specified and the format specification requires more data from a record than the record contains. If NO is specified, the input list and the format specification must not require more characters from a record than the record contains. If this specifier is omitted, the default value is YES.

9.3.5 The CLOSE Statement. The **CLOSE statement** is used to terminate the connection of a particular file to a unit.

Execution of a CLOSE statement that refers to a unit may occur in any program unit of an executable program and need not occur in the same program unit as the execution of an OPEN statement referring to that unit.

Execution of a CLOSE statement specifying a unit that does not exist or has no file connected to it is permitted and affects no file.

After a unit has been disconnected by execution of a CLOSE statement, it may be connected again within the same executable program, either to the same file or to a different file. After a named file has been disconnected by execution of a CLOSE statement, it may be connected again within the same executable program, either to the same unit or to a different unit, provided that the file still exists.

At termination of execution of an executable program for reasons other than an error condition, all units that are connected are closed. Each unit is closed with status KEEP unless the file status prior to termination of execution was SCRATCH, in which case the unit is closed with status DELETE. Note that the effect is as though a CLOSE statement without a STATUS= specifier were executed on each connected unit.

R907 *close-stmt* **is** CLOSE (*close-spec-list*)

R908 *close-spec* **is** [UNIT=] *external-file-unit*
or IOSTAT= *scalar-int-variable*
or ERR= *label*
or STATUS= *scalar-char-expr*

Constraint: If the optional characters UNIT= are omitted from the unit specifier, the unit specifier must be the first item in the *close-spec-list*.

Constraint: ~~Each~~ specifier ~~must not~~ appear more than once in a given *close-stmt*; an *external-file-unit* must be specified.

Constraint: The *label* used in the ERR= specifier must be the statement label of a branch target statement that appears in the same scoping unit as the CLOSE statement.

The *scalar-char-expr* has a limited list of character values. Any trailing blanks are ignored. If a processor is capable of representing letters in both upper and lower case, the value specified is without regard to case.

The IOSTAT= specifier and ERR= specifier are described in 9.4.1.4 and 9.4.1.5, respectively.

An example of a CLOSE statement is:

```
CLOSE (10, STATUS = 'KEEP')
```

9.3.5.1 STATUS= Specifier in the CLOSE Statement. The *scalar-char-expr* must evaluate to KEEP or DELETE. The STATUS= specifier determines the disposition of the file that is connected to the specified unit. KEEP must not be specified for a file whose status prior to execution of a CLOSE statement is SCRATCH. If KEEP is specified for a file that exists, the file continues to exist after the execution of a CLOSE statement. If KEEP is specified for a file that does not exist, the file will not exist after the execution of a CLOSE statement. If DELETE is specified, the file will not exist after the execution of a CLOSE statement. If this specifier is omitted, the default value is KEEP, unless the file status prior to execution of the CLOSE statement is SCRATCH, in which case the default value is DELETE.

9.4 Data Transfer Statements. The **READ statement** is the data transfer input statement. The **WRITE statement** and the **PRINT statement** are the data transfer output statements.

R909 *read-stmt* **is** READ (*io-control-spec-list*) [*input-item-list*]
or READ *format* [, *input-item-list*]

R910 *write-stmt* **is** WRITE (*io-control-spec-list*) [*output-item-list*]

R911 *print-stmt* **is** PRINT *format* [, *output-item-list*]

Examples of data transfer statements are:

```
      READ (6, *) SIZE
      READ 10, A, B
      WRITE (6, 10) A, S, J
      PRINT 10, A, S, J
10 FORMAT (2E16.3, I5)
```

9.4.1 Control Information List. The *io-control-spec-list* is a **control information list** that includes:

(1) A reference to the source or destination of the data to be transferred

(2) Optional specification of editing processes

(3) Optional specification to identify a record

(4) Optional specification of exception handling

(5) Optional return of count of null values

(6) Optional return of status

(7) Optional record advancing specification

(8) Optional return of number of characters read

available le chars? space remaining for non-advancing output

The control information list governs the data transfer.

R912 *io-control-spec* **is** [UNIT=] *io-unit*
or [FMT=] *format*
or [NML=] *namelist-group-name*
or REC= *scalar-int-expr*
or IOSTAT= *scalar-int-variable*
or ERR= *label*
or END= *label*
or NULLS= *scalar-int-variable*
or ADVANCE= *scalar-char-expr*
or SIZE= *scalar-int-expr*
or EOR= *label*

Constraint: An *io-control-spec-list* must contain exactly one *io-unit* and may contain at most one of each of the other specifiers.

Constraint: An END= or a NULLS= specifier must not appear in a *write-stmt*.

Constraint: The *label* used in the ERR=, EOR=, or END= specifier must be the statement label of a branch target statement that appears in the same scoping unit as the data transfer statement.

Constraint: A *namelist-group-name* must not be present if an *input-item-list* or an *output-item-list* is present in the data transfer statement.

Constraint: An *io-control-spec-list* must not contain both a *format* and a *namelist-group-name*.

Constraint: If the optional characters UNIT= are omitted from the unit specifier, the unit specifier must be the first item in the control information list.

Constraint: If the optional characters FMT= are omitted from the format specifier, the format specifier must be the second item in the control information list and the first item must be the unit specifier without the optional characters UNIT=.

Constraint: If the optional characters NML= are omitted from the namelist specifier, the namelist specifier must be the second item in the control information list and the first item must be the unit specifier without the optional characters UNIT=.

Constraint: If the unit specifier specifies an internal file, the *io-control-spec-list* must not contain a REC= specifier ~~or a *namelist-group-name*~~.

Constraint: A NULLS= specifier may be present only in a list-directed input statement.

Constraint: If the REC= specifier is present, an END= specifier must not appear and the *format* must not be an asterisk specifying list-directed input/output.

Constraint: An ADVANCE= and an EOR= specifier may be present only in a formatted sequential READ or WRITE statement that does not contain a namelist specifier or an internal file unit specifier.

Constraint: A SIZE= specifier may be present only in a nonadvancing formatted sequential READ statement that does not contain a namelist specifier or an internal file unit specifier.

If the data transfer statement contains a *format* or *namelist-group-name*, the statement is a **formatted input/output statement**; otherwise, it is an **unformatted input/output statement**.

In a data transfer statement, the variables specified in IOSTAT=, SIZE=, or NULLS= specifiers, if any, must not be associated with one another, nor with any entity in the data transfer input/output list (9.4.2) or *namelist-group-object-list*, nor with a *do-variable* of an *io-implied-do* in the data transfer input/output list.

In a data transfer statement, if a variable specified in an IOSTAT=, SIZE=, or NULLS= specifier is an array element reference, its subscript values must not be affected by the data transfer, the *io-implied-do* processing, or the definition or evaluation of any other specifier in the *io-control-spec-list*.

An example of a READ statement is:

```
READ (IOSTAT = IOS, UNIT = 6, FMT = '(10F8.2)') A, B
```

9.4.1.1 Format Specifier.

R913 *format* **is** *char-expr*
or *label*
or *
or scalar-int-variable

Constraint: The *label* must be the label of a FORMAT statement that appears in the same scoping unit as the statement containing the format specifier.

The *scalar-int-variable* must have been assigned (8.2.4) the statement label of a FORMAT statement that appears in the same scoping unit as the *format*.

The *char-expr* must evaluate to a character object that is a valid format specification (10.1.1 and 10.1.2). Note that *char-expr* includes a character constant.

If *char-expr* is an array, it is treated as if all of the elements of the array were specified in array element order and were concatenated.

If *format* is *, the statement is a **list-directed input/output statement**.

An example in which the format is a character expression is:

```
READ (6, FMT = "(" // CHAR_FMT // ")" )  X, Y, Z
```

where CHAR_FMT is a character variable.

9.4.1.2 Namelist Specifier. The NML= specifier supplies the *namelist-group-name* (5.4). This name identifies a specific collection of data objects on which transfer is to be performed.

If a *namelist-group-name* is present, the statement is a **namelist input/output statement**.

9.4.1.3 Record Number. The REC= specifier specifies the number of the record that is to be read or written and the file must be connected for direct access. If the control information list contains a REC= specifier, the statement is a **direct access input/output statement**; otherwise, it is a **sequential access input/output statement**.

9.4.1.4 Input/Output Status. Execution of an input/output statement containing the IOSTAT= specifier causes the variable specified in the IOSTAT= specifier to become defined:

(1) With a zero value if neither an error condition, an end-of-file condition, nor an end-of-record condition is encountered by the processor,

(2) With a processor-dependent positive integer value if an error condition is encountered,

(3) With a processor-dependent negative integer value if an end-of-file condition is encountered and no error condition is encountered, or

(4) With a processor-dependent negative integer value different from the end-of-file value if an end-of-record condition and no error or end-of-file condition is encountered.

Note that condition (3) may occur only during execution of a sequential input statement and condition (4) may occur only during execution of a nonadvancing input statement.

Consider the example:

```
READ (FMT = "(E8.3)", UNIT=3, IOSTAT = IOSS) X
   !
IF (IOSS < 0) THEN
   !
   ! PERFORM END-OF-FILE PROCESSING ON THE FILE
   ! CONNECTED TO UNIT 3.
   CALL END_PROCESSING
   !
ELSE IF (IOSS > 0) THEN
   !
   ! PERFORM ERROR PROCESSING
   CALL ERROR_PROCESSING
   !
END IF
```

9.4.1.5 Error Branch. If an input/output statement contains an ERR= specifier and the processor encounters an error condition during execution of the statement:

(1) Execution of the input/output statement terminates,

(2) The position of the file specified in the input/output statement becomes indeterminate,

(3) If the input/output statement also contains an IOSTAT= specifier, the variable specified becomes defined with a processor-dependent positive integer value, and

(4) Execution continues with the statement specified in the ERR= specifier.

9.4.1.6 End-of-File Branch. If an input statement contains an END= specifier and the processor encounters an end-of-file condition and encounters no error condition during execution of the statement:

(1) Execution of the READ statement terminates,

(2) If the file specified in the input statement is an external file, it is positioned after the endfile record.

(3) If the input statement also contains an IOSTAT= specifier, the variable specified becomes defined with a processor-dependent negative integer value, and

(4) Execution continues with the statement specified in the END= specifier.

why?

In a WRITE statement, the control information list must not contain an END= specifier.

9.4.1.7 End-of-Record Branch. If an input statement contains an EOR= specifier and the processor encounters an end-of-record condition during execution of the statement:

(1) If PAD = YES, the record is logically padded to satisfy the input list item and data edit descriptor that requires more characters than the record contains,

(2) Execution of the input statement terminates,

(3) The file specified in the input statement is positioned after the current record,

(4) If the input statement also contains an IOSTAT= specifier, the variable specified becomes defined with a processor-dependent negative integer value, and

(5) Execution continues with the statement specified in the EOR= specifier.

9.4.1.8 Advance Specifier. The *scalar-char-expr* must evaluate to YES or NO. The ADVANCE= specifier determines whether nonadvancing input/output occurs for this input/output statement. If NO is specified, nonadvancing input/output occurs. If YES is specified, normal formatted sequential input/output occurs. If this specifier is omitted, the default value is YES.

9.4.1.9 Character Count. When a ~~nonadvancing~~ input statement terminates, the variable specified in the SIZE= specifier is defined to be the count of the number of characters processed by data edit descriptors, not including logical blank padding. If an error condition or end-of-file condition occurs, the result is processor-dependent.

9.4.1.10 Nulls Count. A **null value** is a value that has no effect on the definition status of the corresponding input list item. If the input list item is defined, it retains its previous value; if it is undefined, it remains undefined. A null value must not be used as either the real or imaginary part of a complex constant, but a single null value may represent an entire complex constant.

When a list-directed input statement terminates, the variable specified in the NULLS= specifier is defined to be the count of the null values read by the statement.

9.4.2 Data Transfer Input/Output List. An input/output list specifies the entities whose values are transferred by a data transfer input/output statement.

R914	*input-item*	**is** *variable*	
		or *io-implied-do*	
R915	*output-item*	**is** *expr*	
		or *io-implied-do*	
R916	*io-implied-do*	**is** (*io-implied-do-object-list* , *io-implied-do-control*)	
R917	*io-implied-do-object*	**is** *input-item*	
		or *output-item*	
R918	*io-implied-do-control*	**is** *do-variable* = *scalar-numeric-expr* , ■	
		■ *scalar-numeric-expr* [, *scalar-numeric-expr*]	

Constraint: The *do-variable* must be a scalar of type integer, default real, or double precision real.

Constraint: In an *input-item-list*, an *io-implied-do-object* must be an *input-item*. In an *output-item-list*, an *io-implied-do-object* must be an *output-item*.

An *input-item* must not appear as, nor be associated with, the *do-variable* of any *io-implied-do* that contains the *input-item*.

If an input item is a pointer, it must be currently associated with a definable target.

The *do-variable* of an *io-implied-do* that is contained within another *io-implied-do* must not appear as, nor be associated with, the *do-variable* of the containing *io-implied-do*.

If an array appears as an input/output list item, it is treated as if the elements, if any, were specified in array element order (6.2.2.2). The name or array section of an assumed-size array must not appear as an input/output list item.

The name of a derived-type object must not appear as an input/output list item if any component ultimately contained within the object is not accessible within the scoping unit containing the input/output statement.

If a derived type contains a pointer component, an object of this type must not appear as an input item nor as the result of the expression evaluation in an input/output list.

If the name of a derived-type object appears as an input/output list item in a formatted input/output statement, it is treated as if all of the components of the object were specified in the same order as in the definition of the derived type.

If the name or subobject designator of a derived-type object appears as an input/output list item in an unformatted input/output statement, it is treated as a single value in a processor-dependent form. Note that, in this case, the appearance of a derived-type object as an input/output list item is not equivalent to the list of its components.

For an implied do, the loop initialization and execution is the same as for a DO construct (8.1.4.4).

Note that a constant, an expression involving operators or function references, or an expression enclosed in parentheses may appear as an output list item but must not appear as an input list item.

If the input item is a pointer, data are transferred from the file into the associated target.

An example of an output list with an implied DO is:

```
WRITE (LP, FMT = '(10F8.2)') (LOG (A (I)), I = 1, N + 9, K), G
```

9.4.3 Error, End-of-Record, and End-of-File Conditions. The set of input/output error conditions is processor dependent.

An **end-of-file condition** exists if either of the following events occurs:

(1) An endfile record is encountered during the reading of a file connected for sequential access.

(2) An attempt is made to read a record beyond the end of an internal file.

Note that an end-of-file condition may occur at the beginning of an input statement or within a formatted input statement when more than one record is required by the interaction of the input/output list and the format.

If an error condition or an end-of-file condition occurs during execution of an input/output statement, execution of the input/output statement terminates and any implied-do variables become undefined. If an error condition occurs during execution of an input/output statement, the position of the file becomes indeterminate.

If an error or end-of-file condition occurs on input, all list items become undefined.

If an end-of-record condition occurs during execution of a nonadvancing input statement, the following occurs: if PAD=YES, the record is logically padded to satisfy the input list item and data edit descriptor that requires more characters than the record contains; execution of the input statement terminates; and the file specified in the input statement is positioned after the current record.

If an error condition occurs during execution of an input/output statement that contains neither an IOSTAT= nor an ERR= specifier, or if an end-of-file condition occurs during execution of a READ statement that contains neither an IOSTAT= specifier nor an END= specifier, or if an end-of-record condition occurs during execution of a nonadvancing READ statement that contains neither an IOSTAT= specifier nor an EOR= specifier, execution of the executable program is terminated.

9.4.4 Execution of a Data Transfer Input/Output Statement. The effect of executing a data transfer input/output statement must be as if the following operations were performed in the order specified:

(1) Determine the direction of data transfer

(2) Identify the unit

(3) Establish the format if one is specified

(4) Position the file prior to data transfer (9.2.1.3.2)

(5) Transfer data between the file and the entities specified by the input/output list (if any) or namelist

(6) Position the file after data transfer (9.2.1.3.3)

(7) Cause the variables specified in the IOSTAT= and NULLS= specifiers, if specified, to become defined.

9.4.4.1 Direction of Data Transfer. Execution of a READ statement causes values to be transferred from a file to the entities specified by the input list, if any, or specified within the file itself for namelist input. Execution of a WRITE or PRINT statement causes values to be transferred to a file from the entities specified by the output list and format specification, if any, or by the *namelist-group-name* for namelist output. Execution of a WRITE or PRINT statement for a file that does not exist creates the file unless an error condition occurs.

9.4.4.2 Identifying a Unit. A data transfer input/output statement that contains an input/output control list includes a unit specifier that identifies an external unit or an internal file. A READ statement that does not contain an input/output control list specifies a particular processor-dependent unit, which is the same as the unit identified by * in a READ statement that contains an input/output control list. The PRINT statement specifies some other processor-dependent unit, which is the same as the unit identified by * in a WRITE statement. Thus, each data transfer input/output statement identifies an external unit or an internal file.

The unit identified by a data transfer input/output statement must be connected to a file when execution of the statement begins. Note that the file may be preconnected.

9.4.4.3 Establishing a Format. If the input/output control list contains * as a format, list-directed formatting is established. If *namelist-group-name* is present, namelist formatting is established. If no *format* or *namelist-group-name* is specified, unformatted data transfer is established. Otherwise, the format specification identified by the format specifier, is established. If the format is an array, the effect is as if all elements of the array were concatenated in array element order.

On output, if an internal file has been specified, a format specification that is in the file or is associated with the file must not be specified.

9.4.4.4 Data Transfer. Data are transferred between records and entities specified by the input/output list or namelist. The list items are processed in the order of the input/output list for all data transfer input/output statements except namelist formatted data transfer input statements. The list items for a namelist formatted data transfer input statement are processed in the order of the entities specified within the input records.

All values needed to determine which entities are specified by an input/output list item are determined at the beginning of the processing of that item.

All values are transmitted to or from the entities specified by a list item prior to the processing of any succeeding list item for all data transfer input/output statements. In the example,

```
READ (N) N, X (N)
```

the old value of N identifies the unit, but the new value of N is the subscript of X.

All values following the *name*= part of the namelist entity (10.9) within the input records are transmitted to the matching entity specified in the *namelist-group-object-list* prior to processing any succeeding entity within the input record for namelist formatted data transfer input statements. If an entity is specified more than once within the input record during a namelist formatted data transfer input statement, the last occurrence of the entity specifies the value or values to be used for that entity. An input list item, or an entity associated with it, must not contain any portion of an established format specification.

If an internal file has been specified, an input/output list item must not be in the file or associated with the file. Note that the file is a character object.

A DO variable becomes defined and its iteration count established at the beginning of processing of the items that constitute the range of an *io-implied-do*.

On output, every entity whose value is to be transferred must be defined.

On input, an attempt to read a record of a file connected for direct access that has not previously been written causes all entities specified by the input list to become undefined.

9.4.4.4.1 Unformatted Data Transfer. During unformatted data transfer, data are transferred without editing between the current record and the entities specified by the input/output list. Exactly one record is read or written.

Objects of intrinsic or derived types may be transferred through an unformatted data transfer statement.

On input, the file must be positioned so that the record read is an unformatted record or an endfile record. The number of values required by the input list must be less than or equal to the number of values in the record.

On input, each value in the record must be of the same type and have the same type parameter values as the corresponding entity in the input list, except that one complex value may correspond to two real list entities or two real values may correspond to one complex list entity. Note that if an entity in the input list is of type character, the character entity must have the same length and the same kind type parameter as the character value.

On output to a file connected for direct access, the output list must not specify more values than can fit into the record. If the file is connected for direct access and the values specified by the output list do not fill the record, the remainder of the record is undefined.

If the file is connected for sequential access, the record is created with a length sufficient to hold the values from the output list. This length must be one of the set of allowed record lengths for the file and must not exceed the value specified in the RECL= specifier, if any, of the OPEN statement that established the connection.

If the file is connected for formatted input/output, unformatted data transfer is prohibited.

The unit specified must be an external unit.

9.4.4.4.2 Formatted Data Transfer. During formatted data transfer, data are transferred with editing between the file and the entities specified by the input/output list or by the *namelist-group-name*, if any. Format control is initiated and editing is performed as described in Section 10. The current record and possibly additional records are read or written.

Values may be transmitted through a formatted data transfer statement to or from objects of intrinsic or derived types. In the latter case, the transmission is in the form of values of intrinsic types to or from the components of intrinsic types which ultimately comprise these structured objects.

On input, the file must be positioned so that the record read is a formatted record or an endfile record.

If the input item is a pointer, data are transferred from the file into the associated target.

If the file is connected for unformatted input/output, formatted data transfer is prohibited.

The input record is logically padded with blanks to satisfy an input list and format specification that requires more characters than the record contains, unless PAD = NO was specified in the OPEN statement. If PAD = NO was specified, the input list and format specification must not require more characters from the record than the record contains.

why does it matter if unformatted to this also? FTT explains.

If the file is connected for direct access, the record number is increased by one as each succeeding record is read or written.

or RECL is specified?

On output, if the file is connected for direct access or is an internal file and the characters specified by the output list and format do not fill a record, blank characters are added to fill the record.

On output, the output list and format specification must not specify more characters for a record than have been specified by a RECL= specifier in the OPEN statement.

9.4.4.5 List-Directed Formatting. If list-directed formatting has been established, editing is performed as described in 10.8.

9.4.4.6 Namelist Formatting. If namelist formatting has been established, editing is performed as described in 10.9.

9.4.5 Printing of Formatted Records. The transfer of information in a formatted record to certain devices determined by the processor is called **printing**. If a formatted record is printed, the first character of the record is not printed. The remaining characters of the record, if any, are printed in one line beginning at the left margin.

The first character of such a record determines vertical spacing as follows:

Character	Vertical Spacing Before Printing
Blank	One Line
0	Two Lines
1	To First Line of Next Page
+	No Advance

If there are no characters in the record, the vertical spacing is one line and no characters other than blank are printed in that line.

The PRINT statement does not imply that printing will occur, and the WRITE statement does not imply that printing will not occur.

9.4.6 Termination of Data Transfer Statements. Termination of an input/output data transfer statement occurs when any of the following conditions are met:

(1) All elements of the *input-item-list* or *output-item-list* have been read or written, with or without editing, from or to the specified file.

(2) An error condition is encountered.

(3) An end-of-file condition is encountered.

(4) A slash (/) is encountered as a value separator (10.8, 10.9) in the record being read during list-directed or namelist input.

(5) An end-of-record condition occurs during a nonadvancing input/output statement.

9.5 File Positioning Statements.

R919 *backspace-stmt* **is** BACKSPACE *external-file-unit*
or BACKSPACE (*position-spec-list*)

R920 *endfile-stmt* **is** ENDFILE *external-file-unit*
or ENDFILE (*position-spec-list*)

R921 *rewind-stmt* **is** REWIND *external-file-unit*
or REWIND (*position-spec-list*)

A file that is not connected for sequential access must not be referred to by a BACKSPACE, an ENDFILE, or a REWIND statement.

R922 *position-spec* **is** [UNIT =] *external-file-unit*
or IOSTAT = *scalar-int-variable*
or ERR = *label*

Constraint: The *label* used in the ERR= specifier must be the statement label of a branch target statement that appears in the same scoping unit as the file positioning statement.

Constraint: If the optional characters UNIT= are omitted from the unit specifier, the unit specifier must be the first item in the *position-spec-list*.

Constraint: A *position-spec-list* must contain exactly one *external-file-unit* and may contain at most one of each of the other specifiers.

The IOSTAT= and ERR= specifiers are described in 9.4.1.4 and 9.4.1.5, respectively.

9.5.1 BACKSPACE Statement. If there is a current record and the current character position is not at the beginning of the current record, the specified unit is positioned before the current record. Otherwise, execution of a BACKSPACE statement causes the file connected to the specified unit to be positioned before the current record if there is a current record, or before the preceding record if there is no current record. If there is no preceding record, the position of the file is not changed. Note that if the preceding record is an endfile record, the file becomes positioned before the endfile record. If a BACKSPACE statement causes the implicit writing of an endfile record, the file becomes positioned before the record that precedes the endfile record.

Backspacing a file that is connected but does not exist is prohibited.

Backspacing over records written using list-directed or namelist formatting is prohibited.

An example of a BACKSPACE statement is:

```
BACKSPACE (10, ERR = 20)
```

9.5.2 ENDFILE Statement. Execution of an ENDFILE statement writes an endfile record ~~which becomes the last record of the file~~ as the next record of the file. The file is then positioned after the endfile record which becomes the last record of the file. If the file may also be connected for direct access, only those records before the endfile record are considered to have been written. Thus, only those records may be read during subsequent direct access connections to the file.

After execution of an ENDFILE statement, a BACKSPACE or REWIND statement must be used to reposition the file prior to execution of any data transfer input/output statement.

Execution of an ENDFILE statement for a file that is connected but does not exist creates the file prior to writing the endfile record.

An example of an ENDFILE statement is:

```
ENDFILE K
```

9.5.3 REWIND Statement. Execution of a REWIND statement causes the specified file to be positioned at its initial point. Note that if the file is already positioned at its initial point, execution of this statement has no effect on the position of the file.

Execution of a REWIND statement for a file that is connected but does not exist is permitted and has no effect.

An example of a REWIND statement is:

```
REWIND 10
```

9.6 File Inquiry. The INQUIRE statement may be used to inquire about properties of a particular named file or of the connection to a particular unit. There are three forms of the INQUIRE statement: **inquire by file**, which uses the FILE= specifier, **inquire by unit**, which uses the UNIT= specifier, and **inquire by output list**, which uses only the IOLENGTH= specifier. All specifier value assignments are performed according to the rules for assignment statements.

An INQUIRE statement may be executed before, while, or after a file is connected to a unit. All values assigned by an INQUIRE statement are those that are current at the time the statement is executed.

R923 *inquire-stmt* **is** INQUIRE (*inquire-spec-list*)
or INQUIRE (IOLENGTH = *scalar-int-variable*) *output-item-list*

Examples of INQUIRE statements are:

```
INQUIRE (IOLENGTH = IOL) A (1:N)
INQUIRE (UNIT = JOAN, OPENED = LOG_01, NAMED = LOG_02, &
   FORM = CHAR_VAR, IOSTAT = IOS)
```

9.6.1 Inquiry Specifiers. Unless constrained, the following inquiry specifiers may be used in either of the inquire by file or inquire by unit forms of the INQUIRE statement:

R924 *inquire-spec* **is** [UNIT =] *external-file-unit*
or FILE = *file-name-expr*
or IOSTAT = *scalar-int-variable*
or ERR = *label*
or EXIST = *scalar-logical-variable*
or OPENED = *scalar-logical-variable*
or NUMBER = *scalar-int-variable*
or NAMED = *scalar-logical-variable*
or NAME = *scalar-char-variable*
or ACCESS = *scalar-char-variable*
or SEQUENTIAL = *scalar-char-variable*
or DIRECT = *scalar-char-variable*
or FORM = *scalar-char-variable*
or FORMATTED = *scalar-char-variable*
or UNFORMATTED = *scalar-char-variable*
or RECL = *scalar-int-variable*
or NEXTREC = *scalar-int-variable*
or BLANK = *scalar-char-variable*
or POSITION = *scalar-char-variable*
or ACTION = *scalar-char-variable*
or READ = *scalar-char-variable*
or WRITE = *scalar-char-variable*
or READWRITE = *scalar-char-variable*
or DELIM = *scalar-char-variable*
or PAD = *scalar-char-variable*

Constraint: An *inquire-spec-list* must contain one FILE= specifier or one UNIT= specifier, but not both, and at most one of each of the other specifiers.

Constraint: In the inquire by unit form of the INQUIRE statement, if the optional characters UNIT= are omitted from the unit specifier, the unit specifier must be the first item in the *inquire-spec-list*.

When a returned value of a specifier other than the NAME= specifier is of type character and the processor is capable of representing letters in both upper and lower case, the value returned is in upper case.

If an error condition occurs during execution of an INQUIRE statement, all of the inquiry specifier variables become undefined, except for the variable in the IOSTAT= specifier (if any).

The IOSTAT= and ERR= specifiers are described in 9.4.1.4 and 9.4.1.5, respectively.

9.6.1.1 FILE= Specifier in the INQUIRE Statement. The value of the *file-name-expr* in the FILE= specifier specifies the name of the file being inquired about. The named file need not exist or be connected to a unit. The value of the *file-name-expr* must be of a form acceptable to the processor as a file name. If a processor is capable of representing letters in both upper and lower case, the interpretation of case is processor dependent.

9.6.1.2 EXIST= Specifier in the INQUIRE Statement. Execution of an INQUIRE by file statement causes the *scalar-logical-variable* in the EXIST= specifier to be assigned the value true if there exists a file with the specified name; otherwise, false is assigned. Execution of an INQUIRE by unit statement causes true to be assigned if the specified unit exists; otherwise, false is assigned.

9.6.1.3 OPENED= Specifier in the INQUIRE Statement. Execution of an INQUIRE by file statement causes the *scalar-logical-variable* in the OPENED= specifier to be assigned the value true if the file specified is connected to a unit; otherwise, false is assigned. Execution of an INQUIRE by unit statement causes the *scalar-logical-variable* to be assigned the value true if the specified unit is connected to a file; otherwise, false is assigned.

9.6.1.4 NUMBER= Specifier in the INQUIRE Statement. The *scalar-int-variable* in the NUMBER= specifier is assigned the value of the external unit identifier of the unit that is currently connected to the file. If there is no unit connected to the file, the value –1 is assigned.

9.6.1.5 NAMED= Specifier in the INQUIRE Statement. The *scalar-logical-variable* in the NAMED= specifier is assigned the value true if the file has a name; otherwise, it is assigned the value false.

9.6.1.6 NAME= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the NAME= specifier is assigned the value of the name of the file if the file has a name; otherwise, it becomes undefined. Note that if this specifier appears in an INQUIRE by file statement, its value is not necessarily the same as the name given in the FILE= specifier. For example, the processor may return a file name qualified by a user identification. However, the value returned must be suitable for use as the value of the *file-name-expr* in the FILE= specifier in an OPEN statement. If a processor is capable of representing letters in both upper and lower case, the interpretation of case is processor dependent.

9.6.1.7 ACCESS= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the ACCESS= specifier is assigned the value SEQUENTIAL if the file is connected for sequential access, and DIRECT if the file is connected for direct access. If there is no connection, it is assigned the value UNDEFINED.

9.6.1.8 SEQUENTIAL= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the SEQUENTIAL= specifier is assigned the value YES if SEQUENTIAL is included in the set of allowed access methods for the file, NO if SEQUENTIAL is not included in the set of allowed access methods for the file, and UNKNOWN if the processor is unable to determine whether or not SEQUENTIAL is included in the set of allowed access methods for the file.

9.6.1.9 DIRECT= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the DIRECT= specifier is assigned the value YES if DIRECT is included in the set of allowed access methods for the file, NO if DIRECT is not included in the set of allowed access methods for the file, and UNKNOWN if the processor is unable to determine whether or not DIRECT is included in the set of allowed access methods for the file.

9.6.1.10 FORM= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the FORM= specifier is assigned the value FORMATTED if the file is connected for formatted input/output, and is assigned the value UNFORMATTED if the file is connected for unformatted input/output. If there is no connection, it is assigned the value UNDEFINED.

9.6.1.11 FORMATTED= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the FORMATTED= specifier is assigned the value YES if FORMATTED is included in the set of allowed forms for the file, NO if FORMATTED is not included in the set of allowed forms for the file, and UNKNOWN if the processor is unable to determine whether or not FORMATTED is included in the set of allowed forms for the file.

9.6.1.12 UNFORMATTED= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the UNFORMATTED= specifier is assigned the value YES if UNFORMATTED is included in the set of allowed forms for the file, NO if UNFORMATTED is not included in the set of allowed forms for the file, and UNKNOWN if the processor is unable to determine whether or not UNFORMATTED is included in the set of allowed forms for the file.

9.6.1.13 RECL= Specifier in the INQUIRE Statement. The *scalar-int-variable* in the RECL= specifier is assigned the value of the maximum record length of the file. If the file is connected for formatted input/output, the length is the number of characters. If the file is connected for unformatted input/output, the length is measured in processor-dependent units. If the file does not exist, the *scalar-int-variable* becomes undefined.

9.6.1.14 NEXTREC= Specifier in the INQUIRE Statement. The *scalar-int-variable* in the NEXTREC= specifier is assigned the value $n+1$, where n is the record number of the last record read or written on the file connected for direct access. If the file is connected but no records have been read or written since the connection, the *scalar-int-variable* is assigned the value 1. If the file is not connected for direct access or if the position of the file is indeterminate because of a previous error condition, the *scalar-int-variable* becomes undefined.

9.6.1.15 BLANK= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the BLANK= specifier is assigned the value NULL if null blank control is in effect for the file connected for formatted input/output, and is assigned the value ZERO if zero blank control is in effect for the file connected for formatted input/output. If there is no connection, or if the connection is not for formatted input/output, the *scalar-char-variable* is assigned the value UNDEFINED.

9.6.1.16 POSITION= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the POSITION= specifier is assigned the value REWIND if the file is connected by an OPEN statement for positioning at its initial point, APPEND if the file is connected for positioning before its endfile record or at its terminal point, and ASIS if the file is connected without changing its position. If there is no connection or if the file is connected for direct access, the *scalar-char-variable* is assigned the value UNDEFINED. If the file has been repositioned since the connection, the *scalar-char-variable* is assigned the value UNDEFINED.

9.6.1.17 ACTION= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the ACTION= specifier is assigned the value READ if the file is connected for input only, WRITE if the file is connected for output only, and READWRITE if it is connected for both input and output. If there is no connection, the *scalar-char-variable* is assigned the value UNDEFINED.

9.6.1.18 READ= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the READ= specifier is assigned the value YES if READ is included in the set of allowed actions for the file, NO if READ is not included in the set of allowed actions for the file, and UNKNOWN if the processor is unable to determine whether or not READ is included in the set of allowed actions for the file.

9.6.1.19 WRITE= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the WRITE= specifier is assigned the value YES if WRITE is included in the set of allowed actions for the file, NO if WRITE is not included in the set of allowed actions for the file, and UNKNOWN if the processor is unable to determine whether or not WRITE is included in the set of allowed actions for the file.

9.6.1.20 READWRITE= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the READWRITE= specifier is assigned the value YES if READWRITE is included in the set of allowed actions for the file, NO if READWRITE is not included in the set of allowed actions for the file, and UNKNOWN if the processor is unable to determine whether or not READWRITE is included in the set of allowed actions for the file.

9.6.1.21 DELIM= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the DELIM= specifier is assigned the value APOSTROPHE if the apostrophe is to be used to delimit character data written by list-directed or namelist formatting. If the quotation mark is used to delimit such data, the value QUOTE is assigned. If neither the apostrophe nor the quote is used to delimit the character data, the value NONE is assigned. If there is no connection or if the connection is not for formatted input/output, the *scalar-char-variable* is assigned the value UNDEFINED.

9.6.1.22 PAD= Specifier in the INQUIRE Statement. The *scalar-char-variable* in the PAD= specifier is assigned the value NO if the connection of the file to the unit included the PAD= specifier and its value was NO. Otherwise, the *scalar-char-variable* is assigned the value YES.

9.6.2 Restrictions on Inquiry Specifiers. A variable that may become defined or undefined as a result of its use in a specifier in an INQUIRE statement, or any associated entity, must not appear in another specifier in the same INQUIRE statement.

The *inquire-spec-list* in an INQUIRE by file statement must contain exactly one FILE= specifier and must not contain a UNIT= specifier. The *inquire-spec-list* in an INQUIRE by unit statement must contain exactly one UNIT= specifier and must not contain a FILE= specifier. The unit specified need not exist or be connected to a file. If it is connected to a file, the inquiry is being made about the connection and about the file connected.

9.6.3 IOLENGTH= Specifier in the INQUIRE Statement. The *scalar-int-variable* in the IOLENGTH= specifier is assigned the processor-dependent value that would result from the use of the output list in an unformatted output statement. The value must be suitable as a RECL= specifier in an OPEN statement that connects a file for unformatted direct access when there are input/output statements with the same input/output list.

9.7 Restrictions on Function References and List Items. A function reference must not appear in an expression anywhere in an input/output statement if such a reference causes another input/output statement to be executed. Note that restrictions in the evaluation of expressions (7.1.7) prohibit certain side effects.

9.8 Restriction on Input/Output Statements. If a unit, or a file connected to a unit, does not have all of the properties required for the execution of certain input/output statements, those statements must not refer to the unit.

10. INPUT/OUTPUT EDITING

A format used in conjunction with an input/output statement provides information that directs the editing between the internal representation of data and the character strings of a record or a sequence of records in a file.

A format specifier (9.4.1.1) in an input/output statement may refer to a FORMAT statement or to a character expression that contains a format specification. A format specification provides explicit editing information. The format specifier also may be an asterisk (*) which indicates list-directed formatting (10.8). Instead of a format specifier, a *namelist-group-name* may be specified which indicates namelist formatting (10.9).

10.1 Explicit Format Specification Methods. Explicit format specification may be given:

(1) In a FORMAT statement, or

(2) As the value of a character expression.

10.1.1 FORMAT Statement.

R1001 *format-stmt* **is** FORMAT *format-specification*

R1002 *format-specification* **is** ([*format-item-list*])

Constraint: The *format-stmt* must be labeled.

Constraint: The comma used to separate *format-items* in a *format-item-list* may optionally be omitted as follows:

(1) Between a P edit descriptor and an immediately following F, E, EN, D, or G edit descriptor (10.6.5)

(2) Before a slash edit descriptor when the optional repeat specification is not present (10.6.2)

(3) After a slash edit descriptor

(4) Before or after a colon edit descriptor (10.6.3)

Blank characters may appear at any point within the format specification, with no effect on the format specification, except within a character string edit descriptor (10.7.1, 10.7.2).

Examples of FORMAT statements are:

```
5     FORMAT (1PE12.4, I10)
9     FORMAT (I12, /, ' Dates: ', 2 (2I3, I5))
```

10.1.2 Character Format Specification. A character expression used as a format specifier in a formatted input/output statement must evaluate to a character string whose value constitutes a valid format specification. Note that the format specification begins with a left parenthesis and ends with a right parenthesis.

All character positions up to and including the final right parenthesis of the format specification must be defined at the time the input/output statement is executed, and must not become redefined or undefined during the execution of the statement. Character positions, if any, following the right parenthesis that ends the format specification need not be defined and may contain any character data with no effect on the format specification.

If the format specifier identifies a character array entity, it is treated as if all of the elements of the array were specified in array element order and were concatenated. However, if a format specifier refers to a character array element, the format specification must be contained entirely within that array element.

10.2 Form of a Format Item List.

R1003 *format-item* **is** [*r*] *data-edit-desc*
or *control-edit-desc*
or *char-string-edit-desc*
or [*r*] (*format-item-list*)

R1004 *r* **is** *int-literal-constant*

Constraint: *r* must be positive.

The integer literal constant *r* is called a **repeat specification**.

10.2.1 Edit Descriptors. An **edit descriptor** is used to specify the form of a record and to direct the editing between the characters in a record and internal representations of data. The internal representation of a datum corresponds to the internal representation of a constant of the corresponding type.

An edit descriptor is either a **data edit descriptor**, **control edit descriptor**, or **character string edit descriptor**.

R1005 *data-edit-desc* **is** I *w* [. *m*]
or B *w* [. *m*]
or O *w* [. *m*]
or Z *w* [. *m*]
or F *w* . *d*
or E *w* . *d* [E *e*]
or EN *w* . *d* [E *e*]
or G *w* . *d* [E *e*]
or L *w*
or A [*w*]
or D *w* . *d*
or B *w*
or O *w*
or Z *w*

R1006 *w* **is** *int-literal-constant*

R1007 *m* **is** *int-literal-constant*

R1008 *d* **is** *int-literal-constant*

R1009 *e* **is** *int-literal-constant*

Constraint: *w* and *e* must be positive and *d* and *m* must be zero or positive.

The value of *m*, *d*, and *e* may be restricted further by the value of *w*. I, B, O, Z, F, E, EN, G, L, A, and D indicate the manner of editing.

R1010 *control-edit-desc* **is** *position-edit-desc*
or [*r*] /
or :
or *sign-edit-desc*
or *k* P
or *blank-interp-edit-desc*

R1011 *k* **is** *signed-int-literal-constant*

R1012 *position-edit-desc* **is** T *n*
or TL *n*
or TR *n*
or *n* X

R1013 *n* **is** *int-literal-constant*

Constraint: *n* must be positive.

R1014 *sign-edit-desc* **is** S
or SP
or SS

R1015 *blank-interp-edit-desc* **is** BN
or BZ

In *k*P, *k* is called the **scale factor.**

T, TL, TR, X, slash, colon, S, SP, SS, P, BN, and BZ indicate the manner of editing.

R1016 *char-string-edit-desc* **is** *char-literal-constant*
or *c* H *rep-char* [*rep-char*]...

R1017 *c* **is** *int-literal-constant*

Constraint: *c* must be positive.

Each *rep-char* in a character string edit descriptor must be one of the characters capable of representation by the processor.

The character string edit descriptors provide constant data to be output, and are not valid for input.

Within a character 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.

In the H edit descriptor, *c* specifies the number of characters following the H that comprise the descriptor.

If a processor is capable of representing letters in both upper and lower case, the edit descriptors are without regard to case except for the characters following the H in the H edit descriptor and the character constants.

10.2.2 Fields. A **field** is a part of a record that is read on input or written on output when format control encounters a data edit descriptor or a character string edit descriptor. The **field width** is the size in characters of the field.

10.3 Interaction Between Input/Output List and Format. The beginning of formatted data transfer using a format specification initiates **format control**. Each action of format control depends on information jointly provided by:

(1) The next edit descriptor contained in the format specification, and

(2) The next effective item in the input/output list, if one exists. Zero-sized arrays, zero-sized array sections, zero-length character entities, and implied-DO lists with iteration counts of zero are ignored in determining the next effective item.

If an input/output list specifies at least one list item, at least one data edit descriptor must exist in the format specification. Note that an empty format specification of the form ( ) may be used only if the input/output list is empty or each item is of zero size or length; in this case, one input record is skipped or one output record containing no characters is written.

Except for a format item preceded by a repeat specification *r*, a format specification is interpreted from left to right.

A format item preceded by a repeat specification is processed as a list of *r* items, each identical to the format item but without the repeat specification and separated by commas. Note that an omitted repeat specification is treated in the same way as a repeat specification whose value is one.

To each data edit descriptor interpreted in a format specification, there corresponds one effective item specified by the input/output list (9.4.2), except that an input/output list item of type complex requires the interpretation of two F, E, EN, D, or G edit descriptors. For each control edit

descriptor or character edit descriptor, there is no corresponding item specified by the input/output list, and format control communicates information directly with the record.

Whenever format control encounters a data edit descriptor in a format specification, it determines whether there is a corresponding effective item specified by the input/output list. If there is such an item, it transmits appropriately edited information between the item and the record, and then format control proceeds. If there is no such item, format control terminates.

If format control encounters a colon edit descriptor in a format specification and another effective input/output list item is not specified, format control terminates.

If format control encounters the rightmost parenthesis of a complete format specification and another effective input/output list item is not specified, format control terminates. However, if another effective input/output list item is specified, the file is positioned in a manner identical to the way it is positioned when a slash edit descriptor is processed (10.6.2). Format control then reverts to the beginning of the format item terminated by the last preceding right parenthesis. If there is no such preceding right parenthesis, format control reverts to the first left parenthesis of the format specification. If any reversion occurs, the reused portion of the format specification must contain at least one data edit descriptor. If format control reverts to a parenthesis that is preceded by a repeat specification, the repeat specification is reused. Reversion of format control, of itself, has no effect on the scale factor (10.6.5.1), the sign control edit descriptors (10.6.4), or the blank interpretation edit descriptors (10.6.6).

Example: The format specification:

```
10 FORMAT (1X, 2(F10.3, I5))
```

with an output list of

```
WRITE (10,10) 10.1, 3, 4.7, 1, 12.4, 5, 5.2, 6
```

produces the same output as the format specification:

```
10 FORMAT (1X, F10.3, I5, F10.3, I5/F10.3, I5, F10.3, I5)
```

10.4 Positioning by Format Control. After each data edit descriptor or character string edit descriptor is processed, the file is positioned after the last character read or written in the current record.

After each T, TL, TR, or X, edit descriptor is processed, the file is positioned as described in 10.6.1. After each slash edit descriptor is processed, the file is positioned as described in 10.6.2.

If format control reverts as described in 10.3, the file is positioned in a manner identical to the way it is positioned when a slash edit descriptor is processed (10.6.2).

During a read operation, any unprocessed characters of the current record are skipped whenever the next record is read.

10.5 Data Edit Descriptors. Data edit descriptors cause the conversion of data to or from its internal representation. On input, the specified variable becomes defined unless there is an error or an end-of-file condition occurs. On output, the specified expression is evaluated.

10.5.1 Numeric Editing. The I, F, E, EN, D, and G edit descriptors are used to specify the input/output of integer, real, and complex data. The following general rules apply:

(1) On input, leading blanks are not significant. The interpretation of blanks, other than leading blanks, is determined by a combination of any BLANK= specifier (9.3.4.6), the default for a preconnected or internal file, and any BN or BZ blank control that is currently in effect for the unit (10.6.6). Plus signs may be omitted. A field containing only blanks is considered to be zero.

(2) On input, with F, E, EN, and D, editing, a decimal point appearing in the input field overrides the portion of an edit descriptor that specifies the decimal point location. The

input field may have more digits than the processor uses to approximate the value of the datum.

(3) On output, the representation of a positive or zero internal value in the field may be prefixed with a plus, as controlled by the S, SP, and SS edit descriptors or the processor. The representation of a negative internal value in the field must be prefixed with a minus. However, the processor must not produce a negative signed zero in a formatted output record.

(4) On output, the representation is right-justified in the field. If the number of characters produced by the editing is smaller than the field width, leading blanks are inserted in the field.

(5) On output, if the number of characters produced exceeds the field width or if an exponent exceeds its specified length using the E*w.d*E*e*, EN*w.d*E*e*, or G*w.d*E*e* edit descriptor, the processor must fill the entire field of width *w* with asterisks. However, the processor must not produce asterisks if the field width is not exceeded when optional characters are omitted. Note that when an SP edit descriptor is in effect, a plus is not optional.

Integer only?

10.5.1.1 Integer Editing. The I*w*, I*w.m*, G*w.d*, G*w.d*E*e*, B*w*, B*w.m*, O*w*, O*w.m*, Z*w*, and Z*w.m* edit descriptors indicate that the field to be edited occupies *w* positions. The specified input/output list item must be of type integer.

On input, *m* has no effect.

In the input field, the character string must be in the form of an optionally signed integer constant, for the I edit descriptor, except for the interpretation of blanks. For the B, O, and Z edit descriptors, the character string must consist of binary, octal, or hexadecimal digits (R408, R409, R410) in the respective input field.

The output field for the I*w* edit descriptor consists of zero or more leading blanks followed by a minus if the value of the internal datum is negative, or an optional plus otherwise, followed by the magnitude of the internal value in the form of an unsigned integer constant without leading zeros. Note that an integer constant always consists of at least one digit.

The input field for the B*w*, O*w*, and Z*w* descriptors consists of zero or more leading blanks followed by the internal value in a form identical to the digits of a binary, octal, or hexadecimal constant, respectively, with the same value.

The output field for the I*w.m*, B*w.m*, O*w.m*, and Z*w.m* edit descriptor is the same as for the I*w*, B*w*, O*w*, and Z*w* edit descriptor, respectively, except that the unsigned integer constant consists of at least *m* digits. If necessary, sufficient leading zeros are included to achieve the minimum of *m* digits. The value of *m* must not exceed the value of *w*. If *m* is zero and the value of the internal datum is zero, the output field consists of only blank characters, regardless of the sign control in effect.

10.5.1.2 Real and Complex Editing. The F, E, EN, and D, edit descriptors specify the editing of real and complex data. An input/output list item corresponding to an F, E, EN, and D, edit descriptor must be real or complex.

10.5.1.2.1 F Editing. The F*w.d* edit descriptor indicates that the field occupies *w* positions, the fractional part of which consists of *d* digits.

The input field consists of an optional sign, followed by a string of digits optionally containing a decimal point, including any blanks interpreted as zeros. The *d* has no effect on input if the input field contains a decimal point. If the decimal point is omitted, the rightmost *d* digits of the string, with leading zeros assumed if necessary, are interpreted as the fractional part of the value represented. The string of digits may contain more digits than a processor uses to approximate the value of the constant. The basic form may be followed by an exponent of one of the following forms:

(1) Explicitly signed integer constant

(2) E followed by zero or more blanks, followed by an optionally signed integer constant

(3) D followed by zero or more blanks, followed by an optionally signed integer constant

An exponent containing a D is processed identically to an exponent containing an E.

Note that if the input field does not contain an exponent, the effect is as if the basic form were followed by an exponent with a value of $-k$, where k is the established scale factor (10.6.5.1).

The output field consists of blanks, if necessary, followed by a minus if the internal value is negative, or an optional plus otherwise, followed by a string of digits that contains a decimal point and represents the magnitude of the internal value, as modified by the established scale factor and rounded to d fractional digits. Leading zeros are not permitted except for an optional zero immediately to the left of the decimal point if the magnitude of the value in the output field is less than one. The optional zero must appear if there would otherwise be no digits in the output field.

10.5.1.2.2 E and D Editing. The E*w.d*, and E*w.d*E*e* edit descriptors indicate that the external field occupies *w* positions, the fractional part of which consists of *d* digits, unless a scale factor greater than one is in effect, and the exponent part consists of *e* digits. The *e* has no effect on input and *d* has no effect on input if the input field contains a decimal point.

The form and interpretation of the input field is the same as for F editing (10.5.1.2.1).

The form of the output field for a scale factor of zero is:

$$[\pm][0].x_1x_2\cdots x_d\,exp$$

where:

$\pm$ signifies a plus or a minus.

$x_1x_2\cdots x_d$ are the d most significant digits of the datum value after rounding.

exp is a decimal exponent having one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
E*w.d*	$\|exp\| \le 99$	$E\pm z_1z_2$ *or* $\pm 0z_1z_2$
	$99 < \|exp\| \le 999$	$\pm z_1z_2z_3$
E*w.d*E*e*	$\|exp\| \le 10^e - 1$	$E\pm z_1z_2\cdots z_e$
D*w.d*	$\|exp\| \le 99$	$D\pm z_1z_2$ or $E\pm z_1z_2$ or $\pm 0z_1z_2$
	$99 < \|exp\| \le 999$	$\pm z_1z_2z_3$

where each z is a digit.

The sign in the exponent is produced. A plus sign is produced if the exponent value is zero. The forms E*w.d* and *Dw.d* must not be used if $|exp| > 999$.

The scale factor k controls the decimal normalization (10.2.1, 10.6.5.1). If $-d < k \le 0$, the output field contains exactly $|k|$ leading zeros and $d - |k|$ significant digits after the decimal point. If

$0 < k < d+2$, the output field contains exactly k significant digits to the left of the decimal point and $d-k+1$ significant digits to the right of the decimal point. Other values of k are not permitted.

10.5.1.2.3 EN Editing. The EN edit descriptor produces an output field in the form of a real number in engineering notation such that the decimal exponent is divisible by three and the absolute value of the mantissa is greater than or equal to one and less than 1000, except when the output value is zero. The scale factor has no effect on output.

The forms of the edit descriptor are EN*w.d* and EN*w.d*E*e* indicating that the external field occupies *w* positions, the fractional part of which consists of *d* digits and the exponent part consists of *e* digits.

The form and interpretation of the input field is the same as for F editing (10.5.1.2.1).

The form of the output field is:

$$[\pm]\, yyy.x_1x_2\cdots x_d\, exp$$

where:

± signifies a plus or a minus.

yyy are the 1 to 3 decimal digits representative of the most significant digits of the value of the datum after rounding (*yyy* is an integer such that $1 \le yyy \le 999$ or $yyy = 0$).

$x_1x_2\cdots x_d$ are the *d* next most significant digits of the value of the datum after rounding.

exp is a decimal exponent, divisible by three, of one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
EN*w.d*	$\lvert exp \rvert \le 99$	$E\pm z_1z_2$ *or* $\pm 0z_1z_2$
	$99 < \lvert exp \rvert \le 999$	$\pm z_1z_2z_3$
EN*w.d*E*e*	$\lvert exp \rvert \le 10^e - 1$	$E\pm z_1z_2\cdots z_e$

where each *z* is a digit.

The sign in the exponent is produced. A plus sign is produced if the exponent value is zero. The form EN*w.d* must not be used if $|exp| > 999$.

Examples:

Internal Value	Output field Using SS, EN12.3
6.421	6.421E+00
–.5	–500.000E–03
.00217	2.170E–03
4721.3	4.721E+03

10.5.1.2.4 Complex Editing. A complex datum consists of a pair of separate real data; therefore, the editing is specified by two edit descriptors each of which specifies the editing of real data. The first of the edit descriptors specifies the real part; the second specifies the imaginary part. The two edit descriptors may be different. Control and character string edit descriptors may be processed between the edit descriptor for the real part and the edit descriptor for the imaginary

part.

10.5.2 Logical Editing. The L*w* edit descriptor indicates that the field occupies *w* positions. The specified input/output list item must be of type logical.

The input field consists of optional blanks, optionally followed by a decimal point, followed by a T for true or F for false. The T or F may be followed by additional characters in the field. Note that the logical constants .TRUE. and .FALSE. are acceptable input forms.

The output field consists of $w - 1$ blanks followed by a T or F, depending on whether the value of the internal datum is true or false, respectively.

10.5.3 Character Editing. The A[*w*] edit descriptor is used with an input/output list item of type character.

If a field width *w* is specified with the A edit descriptor, the field consists of *w* characters. If a field width *w* is not specified with the A edit descriptor, the number of characters in the field is the length of the character input/output list item.

Let *len* be the length of the input/output list item. If the specified field width *w* for A input is greater than or equal to *len*, the rightmost *len* characters will be taken from the input field. If the specified field width *w* is less than *len*, the *w* characters will appear left-justified with $len - w$ trailing blanks in the internal representation.

If the specified field width *w* for A output is greater than *len*, the output field will consist of $w - len$ blanks followed by the *len* characters from the internal representation. If the specified field width *w* is less than or equal to *len*, the output field will consist of the leftmost *w* characters from the internal representation.

10.5.4 Generalized Editing. The G*w.d* and G*w.d*E*e* edit descriptors are used with an input/output list item of any intrinsic type. These edit descriptors indicate that the external field occupies *w* positions, the fractional part of which consists of a maximum of *d* digits and the exponent part consists of *e* digits. When these edit descriptors are used to specify the input/output of integer, logical, or character data, *d* and *e* have no effect.

10.5.4.1 Generalized Numeric Editing. When used to specify the input/output of integer, real, and complex data, the G*w.d* and G*w.d*E*e* edit descriptors follow the general rules for numeric editing (10.5.1). Note that the G*w.d*E*e* edit descriptor follows any additional rules for the E*w.d*E*e* edit descriptor.

10.5.4.1.1 Generalized Integer Editing. When used to specify the input/output of integer data, the G*w.d* and G*w.d*E*e* edit descriptors follow the rules for the I*w* edit descriptor (10.5.1.1).

10.5.4.1.2 Generalized Real and Complex Editing. The form and interpretation of the input field is the same as for F editing (10.5.1.2.1).

The method of representation in the output field depends on the magnitude of the datum being edited. Let *N* be the magnitude of the internal datum. If $0 < N < 0.1$ or $N \geq 10^d$, G*w.d* output editing is the same as *k*PE*w.d* output editing and G*w.d*E*e* output editing is the same as *k*PE*w.d*E*e* output editing, where *k* is the scale factor (10.6.5.1) currently in effect. If $0.1 \leq N < 10^d$ or *N* is identically 0, the scale factor has no effect, and the value of *N* determines the editing as follows:

Magnitude of Datum	Equivalent Conversion
$N=0$	F($w-n$).($d-1$), n('b')
$0.1 \le N < 1$	F($w-n$).d, n('b')
$1 \le N < 10$	F($w-n$).($d-1$), n('b')
.	.
.	.
.	.
$10^{d-2} \le N < 10^{d-1}$	F($w-n$).1, n('b')
$10^{d-1} \le N < 10^{d}$	F($w-n$).0, n('b')

where *b* is a blank. *n* is 4 for G*w.d* and *e* + 2 for G*w.d*E*e*.

Note that the scale factor has no effect unless the magnitude of the datum to be edited is outside of the range that permits effective use of F editing.

10.5.4.2 Generalized Logical Editing. When used to specify the input/output of logical data, the G*w.d* and G*w.d*E*e* edit descriptors follow the rules for the L*w* edit descriptor (10.5.2).

10.5.4.3 Generalized Character Editing. When used to specify the input/ouput of character data, the G*w.d* and G*w.d*E*e* edit descriptors follow the rules for the A*w* edit descriptor (10.5.3).

10.6 Control Edit Descriptors. A control edit descriptor does not cause the transfer of data nor the conversion of data to or from internal representation, but may affect the conversions performed by subsequent data edit descriptors.

10.6.1 Position Editing. The T, TL, TR, and X edit descriptors specify the position at which the next character will be transmitted to or from the record.

The position specified by a T edit descriptor may be in either direction from the current position. On input, this allows portions of a record to be processed more than once, possibly with different editing.

The position specified by an X edit descriptor is forward from the current position. On input, a position beyond the last character of the record may be specified if no characters are transmitted from such positions. Note that an *n*X edit descriptor has the same effect as a TR*n* edit descriptor.

On output, a T, TL, TR, or X edit descriptor does not by itself cause characters to be transmitted and therefore does not by itself affect the length of the record. If characters are transmitted to positions at or after the position specified by a T, TL, TR, or X edit descriptor, positions skipped and not previously filled are filled with blanks. The result is as if the entire record were initially filled with blanks.

On output, a character in the record may be replaced. However, a T, TL, TR, or X edit descriptor never directly causes a character already placed in the record to be replaced. Such edit descriptors may result in positioning such that subsequent editing causes a replacement.

10.6.1.1 T, TL, and TR Editing. The T*n* edit descriptor indicates that the transmission of the next character to or from a record is to occur at the *n*th character position of the record, relative to the character position at the beginning of the input/output statement.

The TL*n* edit descriptor indicates that the transmission of the next character to or from the record is to occur at the character position *n* characters backward from the current position. However, if the current position is less than or equal to the position of the *n*th character, relative to the character position of the current record at the beginning of the input/output statement, then the TL*n* edit descriptor indicates that the transmission of the next character to or from the record is to occur at the first character position of the input/output statement.

The TR*n* edit descriptor indicates that the transmission of the next character to or from the record is to occur at the character position *n* characters forward from the current position.

Note that *n* must be specified and must be greater than zero.

10.6.1.2 X Editing. The *n*X edit descriptor indicates that the transmission of the next character to or from a record is to occur at the position *n* characters forward from the current position. Note that the *n* must be specified and must be greater than zero.

10.6.2 Slash Editing. The slash edit descriptor indicates the end of data transfer on the current record.

On input from a file connected for sequential access, the remaining portion of the current record is skipped and the file is positioned at the beginning of the next record. This record becomes the current record. On output to a file connected for sequential access, a new empty record is created following the current record; this new record then becomes the last and current record of the file and the file is positioned at the beginning of this new record.

For a file connected for direct access, the record number is increased by one and the file is positioned at the beginning of the record that has that record number, if there is a such a record, and this record becomes the current record.

Note that a record that contains no characters may be written on output. If the file is an internal file or a file connected for direct access, the record is filled with blank characters. Note also that an entire record may be skipped on input. The repeat specification is optional on the slash edit descriptor. If it is not specified, the default value is one.

10.6.3 Colon Editing. The colon edit descriptor terminates format control if there are no more effective items in the input/output list (9.4.2). The colon edit descriptor has no effect if there are more effective items in the input/output list.

10.6.4 S, SP, and SS Editing. The S, SP, and SS edit descriptors may be used to control optional plus characters in numeric output fields. At the beginning of execution of each formatted output statement, the processor has the option of producing a plus in numeric output fields. If an SP edit descriptor is encountered in a format specification, the processor must produce a plus in any subsequent position that normally contains an optional plus. If an SS edit descriptor is encountered, the processor must not produce a plus in any subsequent position that normally contains an optional plus. If an S edit descriptor is encountered, the option of producing the plus is restored to the processor.

The S, SP, and SS edit descriptors affect only I, F, E, EN, D, G editing during the execution of an output statement. The S, SP, and SS edit descriptors have no effect during the execution of an input statement.

10.6.5 P Editing. The *k*P edit descriptor sets the value of the scale factor to *k*. The scale factor may affect the editing of numeric quantities.

10.6.5.1 Scale Factor. The value of the scale factor is zero at the beginning of execution of each input/output statement. It applies to all subsequently interpreted F, E, EN, D, and G edit descriptors until another P edit descriptor is encountered, and then a new scale factor is established. Note that reversion of format control (10.3) does not affect the established scale factor.

The scale factor *k* affects the appropriate editing in the following manner:

(1) On input, with F, E, EN, D, and G editing (provided that no exponent exists in the field) and F output editing, the scale factor effect is that the externally represented number equals the internally represented number multiplied by 10^k.

(2) On input, with F, E, EN, D, and G editing, the scale factor has no effect if there is an exponent in the field.

(3) On output, with E and D editing, the significand (4.3.1.2) part of the quantity to be produced is multiplied by 10^k and the exponent is reduced by k.

(4) On output, with G editing, the effect of the scale factor is suspended unless the magnitude of the datum to be edited is outside the range that permits the use of F editing. If the use of E editing is required, the scale factor has the same effect as with E output editing.

(5) On output, with EN editing, the scale factor has no effect.

10.6.6 BN and BZ Editing. The BN and BZ edit descriptors may be used to specify the interpretation of blanks, other than leading blanks, in numeric input fields. At the beginning of execution of each formatted input statement, nonleading blank characters from a file connected by an OPEN statement are interpreted as zeros or are ignored, depending on the value of the BLANK= specifier (9.3.4.6) currently in effect for the unit; a preconnected or an internal file is treated as if the file had been opened with BLANK = NULL. If a BN edit descriptor is encountered in a format specification, all nonleading blank characters in succeeding numeric input fields are ignored. The effect of ignoring blanks is to treat the input field as if blanks had been removed, the remaining portion of the field right-justified, and the blanks replaced as leading blanks. However, a field containing only blanks has the value zero. If a BZ edit descriptor is encountered in a format specification, all nonleading blank characters in succeeding numeric input fields are treated as zeros.

The BN and BZ edit descriptors affect only I, B, O, Z, F, E, EN, D, and G editing during execution of an input statement. They have no effect during execution of an output statement.

10.7 Character String Edit Descriptors. A character string edit descriptor must not be used on input.

10.7.1 Character Constant Edit Descriptor. The character constant edit descriptor causes characters to be written from the enclosed characters of the edit descriptor itself, including blanks. Note that a delimiter is either an apostrophe or quote.

For a character constant edit descriptor, the width of the field is the number of characters contained in, but not including, the delimiting characters. Within the field, two consecutive delimiting characters are counted as a single character.

10.7.2 H Editing. The *c*H edit descriptor causes character information to be written from the next *c* characters (including blanks) following the H of the *c*H edit descriptor in the *format-item-list* itself. If a *c*H edit descriptor occurs within a character constant delimited by apostrophes and the *c*H edit descriptor includes an apostrophe, the apostrophe must be represented by two consecutive apostrophes which are counted as one character in specifying *c*. If a *c*H edit descriptor occurs within a character constant delimited by quotes and the *c*H edit descriptor includes a quote, the quote must be represented by two consecutive quotes which are counted as one character in specifying *c*.

10.8 List-Directed Formatting. The characters in one or more list-directed records constitute a sequence of values and value separators. The end of a record has the same effect as a blank character, unless it is within a character constant. Any sequence of two or more consecutive blanks is treated as a single blank, unless it is within a character constant.

Each value is either a null value or one of the forms:

c
*r***c*
*r**

where *c* is a literal constant or a nondelimited character constant and *r* is an unsigned, nonzero, integer literal constant. The *r***c* form is equivalent to *r* successive appearances of the constant *c*,

and the $r*$ form is equivalent to r successive appearances of the null value. Neither of these forms may contain embedded blanks, except where permitted within the constant *c*.

A **value separator** is one of the following:

(1) A comma optionally preceded by one or more contiguous blanks and optionally followed by one or more contiguous blanks,

(2) A slash optionally preceded by one or more contiguous blanks and optionally followed by one or more contiguous blanks, or

(3) One or more contiguous blanks between two nonblank values or following the last nonblank value, where a nonblank value is a constant, an $r*c$ form, or an $r*$ form.

10.8.1 List-Directed Input. Input forms acceptable to edit descriptors for a given type are acceptable for list-directed formatting, except as noted below. The form of the input value must be acceptable for the type of the input list item. Blanks are never used as zeros, and embedded blanks are not permitted in constants, except within character constants and complex constants as specified below. Note that the end of a record has the effect of a blank, except when it appears within a character constant.

When the corresponding input list item is of type real, the input form is that of a numeric input field. A numeric input field is a field suitable for F editing (10.5.1.2.1) that is assumed to have no fractional digits unless a decimal point appears within the field.

When the corresponding list item is of type complex, the input form consists of a left parenthesis followed by an ordered pair of numeric input fields separated by a comma, and followed by a right parenthesis. The first numeric input field is the real part of the complex constant and the second is the imaginary part. Each of the numeric input fields may be preceded or followed by blanks. The end of a record may occur between the real part and the comma or between the comma and the imaginary part. or between (... , ...)

When the corresponding list item is of type logical, the input form must not include slashes, blanks, or commas among the optional characters permitted for L editing.

When the corresponding list item is of type character, the input form consists of a character constant. Character constants may be continued from the end of one record to the beginning of the next record, but the end of record must not occur between a doubled apostrophe in an apostrophe-delimited constant, nor between a doubled quote in a quote-delimited constant. The end of the record does not cause a blank or any other character to become part of the constant. The constant may be continued on as many records as needed. The characters blank, comma, and slash may appear in character constants.

If the corresponding input list item is of type character and:

(1) The character constant does not contain the value separators blank, comma, or slash, and

(2) The character constant does not cross a record boundary, and

(3) The first nonblank character is not a quotation mark or an apostrophe, and

(4) The leading characters are not numeric followed by an asterisk,

the delimiting apostrophes or quotation marks are not required. If the delimiters are omitted, the character constant is terminated by the first blank, comma, slash, or end of record and apostrophes and quotation marks within the datum are not to be doubled.

Let *len* be the length of the list item, and let *w* be the length of the character constant. If *len* is less than or equal to *w*, the leftmost *len* characters of the constant are transmitted to the list item. If *len* is greater than *w*, the constant is transmitted to the leftmost *w* characters of the list item and the remaining $len - w$ characters of the list item are filled with blanks. Note that the effect is as though the constant were assigned to the list item in a character assignment statement (7.5.1.4).

10.8.1.1 Null Values. A null value is specified by having no characters between successive value separators, no characters preceding the first value separator in the first record read by each execution of a list-directed input statement, or the *r** form. Note that the end of a record following any other value separator, with or without separating blanks, does not specify a null value. A null value has no effect on the definition status of the corresponding input list item. A null value must not be used for either the real or imaginary part of a complex constant, but a single null value may represent an entire complex constant.

A slash encountered as a value separator during execution of a list-directed input statement causes termination of execution of that input statement after the assignment of the previous value. If there are additional items in the input list, the effect is as if null values had been supplied for them. Any DO variable in the input list is defined as though enough null values had been supplied for any remaining input list items.

Note that all blanks in a list-directed input record are considered to be part of some value separator except for the following:

(1) Blanks embedded in a character constant

(2) Embedded blanks surrounding the real or imaginary part of a complex constant

(3) Leading blanks in the first record read by each execution of a list-directed input statement, unless immediately followed by a slash or comma

10.8.2 List-Directed Output. The form of the values produced is the same as that required for input, except as noted otherwise. With the exception of nondelimited character constants, the values are separated by (1) one or more blanks or (2) a comma ~~optionally~~ preceded by ~~one~~ zero or more blanks and ~~optionally~~ followed by ~~one~~ or more blanks.

The processor may begin new records as necessary, but, except for complex constants and character constants, the end of a record must not occur within a constant and blanks must not appear within a constant.

Logical output constants are T for the value true and F for the value false.

Integer output constants are produced with the effect of an Iw edit descriptor.

Real constants are produced with the effect of either an F edit descriptor or an E edit descriptor, depending on the magnitude x of the value and a range $10^{d_1} \le x < 10^{d_2}$, where d_1 and d_2 are processor-dependent integers. If the magnitude x is within this range, the constant is produced using 0PFw.d; otherwise, 1PEw.dEe is used.

For numeric output, reasonable processor-dependent integer values of *w*, *d*, and *e* are used for each of the cases involved.

Complex constants are enclosed in parentheses with a comma separating the real and imaginary parts. The end of a record may occur between the comma and the imaginary part only if the entire constant is as long as, or longer than, an entire record. The only embedded blanks permitted within a complex constant are between the comma and the end of a record and one blank at the beginning of the next record.

Character constants produced for a file opened without a DELIM= specifier (9.3.4.9) or with a DELIM= specifier with a value of NONE:

(1) Are not delimited by apostrophes or quotation marks,

(2) Are not preceded or followed by a value separator,

(3) Have each internal apostrophe or quotation mark represented externally by one apostrophe or quotation mark, and

(4) Have a blank character inserted by the processor for carriage control at the beginning of any record that begins with the continuation of a character constant from the preceding record.

Character constants produced for a file opened with a DELIM= specifier with a value of QUOTE are delimited by quotes, are preceded and followed by a value separator, and have each internal quote represented on the external medium by two contiguous quotes.

Character constants produced for a file opened with a DELIM= specifier with a value of APOSTROPHE are delimited by apostrophes, are preceded and followed by a value separator, and have each internal apostrophe represented on the external medium by two contiguous apostrophes.

If two or more successive values in an output record have identical values, the processor has the option of producing a repeated constant of the form *r*∗*c* instead of the sequence of identical values.

Slashes, as value separators, and null values are not produced as output by list-directed formatting.

Except for continuation of delimited character constants, each output record begins with a blank character to provide carriage control when the record is printed.

10.9 Namelist Formatting. The characters in one or more namelist records constitute a sequence of **name-value subsequences**, each of which consists of an object name or a subobject designator followed by an equals and followed by one or more values and value separators. The equals may ~~optionally~~ be preceded or followed by zero, one, or more contiguous blanks. The end of a record has the same effect as a blank character, unless it is within a character constant. Any sequence of two or more consecutive blanks is treated as a single blank, unless it is within a character constant.

The name may be any name in the *namelist-group-object-list* (5.4).

Each value is either a null value or one of the forms:

c
r∗*c*
r∗

where *c* is a literal constant or a nondelimited character constant (10.9.1.3) and *r* is an unsigned, nonzero, integer literal constant. The *r*∗*c* form is equivalent to *r* successive appearances of the constant *c*, and the *r*∗ form is equivalent to *r* successive null values. Neither of these forms may contain embedded blanks, except where permitted within the constant *c*.

A value separator for namelist formatting is the same as for list-directed (10.8) except that a value separator containing a slash must not immediately precede a value.

10.9.1 Namelist Input. Input for a namelist input statement consists of:

(1) Optional blanks,

(2) The character & followed immediately by the same *namelist-group-name* specified in the namelist input statement,

(3) One or more blanks,

(4) A sequence of zero or more name-value subsequences separated by value separators, and

(5) A slash to terminate the namelist input statement.

In each name-value subsequence, the name must be the name of a namelist group object list item with an optional qualification.

If a processor is capable of representing letters in both upper and lower case, a group name or object name is without regard to case.

10.9.1.1 Namelist Group Object Names. Within the input data, each name must correspond to a specific namelist group object name. Subscripts, strides, and substring range expressions used to qualify group object names must be optionally signed integer literal constants. If a namelist group object name is the name of an array, the name in the input record corresponding to it may be either the array name or the name of an element or section of that array, indicated by qualifying the array name with integer constant subscripts, starting points, and ending points. If the namelist group object name is the name of a variable of derived type, the name in the input record may be either the name of the variable or of one of its components, indicated by qualifying the variable name with the appropriate component name. Successive qualifications may be applied as appropriate to the shape and type of the variable represented.

The order of names in the input records need not match the order of the namelist group object items. The input records need not contain all the names of the namelist group object items. The definition status of any names from the *namelist-group-object-list* that do not occur in the input record remains unchanged. The name in the input record may be preceded and followed by ~~one or more~~ blanks but must not contain embedded blanks.

10.9.1.2 Namelist Input Values. The datum *c* is any input value acceptable to format specifications for a given type, except for a restriction on the form of input values corresponding to list items of type logical as specified in 10.9.1.3. The form of the input value must be acceptable for the type of the namelist group object list item. The number and forms of the input values that may follow the equals in a name-value subsequence depend on the shape and type of the object represented by the name in the input record. When the name in the input record is the name of a scalar variable of an intrinsic type, the equals must not be followed by more than one value. Blanks are never used as zeros, and embedded blanks are not permitted in constants except within character constants and complex constants as specified in 10.9.1.3.

When the name in the input record represents an array variable or a variable of derived type, the effect is as if the variable represented were expanded into a sequence of scalar list items of intrinsic data types, in the same way that formatted input/output list items are expanded (9.4.2). Each input value following the equals must then be acceptable to format specifications for the intrinsic type of the list item in the corresponding position in the expanded sequence, except as noted. The number of values following the equals must not exceed the number of list items in the expanded sequence, but may be less; in the latter case, the effect is as if sufficient null values had been appended to match any remaining list items in the expanded sequence. For example, if the name in the input record is the name of an integer array of size 100, at most 100 values, each of which is either a digit string or a null value, may follow the equals; these values would then be assigned to the elements of the array in array element order.

A slash encountered as a value separator during the execution of a namelist input statement causes termination of execution of that input statement after assignment of the previous value. If there are additional items in the namelist group object being transferred, the effect is as if null values had been supplied for them.

10.9.1.3 Namelist Group Object List Items. When the corresponding namelist group object list item is of type real, the input form of the input value is that of a numeric input field. A numeric input field is a field suitable for F editing (10.5.1.2.1) that is assumed to have no fractional digits unless a decimal point appears within the field.

When the corresponding list item is of type complex, the input form of the input value consists of a left parenthesis followed by an ordered pair of numeric input fields separated by a comma and followed by a right parenthesis. The first numeric input field is the real part of the complex constant and the second part is the imaginary part. Each of the numeric input fields may be preceded or followed by blanks. The end of a record may occur between the real part and the comma or between the comma and the imaginary part.

When the corresponding list item is of type logical, the input form of the input value must not include slashes, blanks, or commas among the optional characters permitted for L editing (10.5.2).

When the corresponding list item is of type character, the input form consists of a character constant. Character constants may be continued from the end of one record to the beginning of the next record, but the end of record must not occur between a doubled apostrophe in an apostrophe-delimited constant, nor between a doubled quote in a quote-delimited constant. The end of the record does not cause a blank or any other character to become part of the constant. The constant may be continued on as many records as needed. The characters blank, comma, and slash may appear in character constants.

If the corresponding list item is of type character and:

(1) The character constant does not contain the value separators blank, comma, or slash,

(2) The character constant does not cross a record boundary,

(3) The first nonblank character is not a quotation mark or an apostrophe, and

(4) The leading characters are not numeric followed by an asterisk,

the delimiting apostrophes or quotation marks are not required. If the delimiters are omitted, the character constant is terminated by the first blank, comma, slash, or end of record and apostrophes and quotation marks within the datum are not to be doubled.

Let *len* be the length of the list item, and let *w* be the length of the character constant. If *len* is less than or equal to *w*, the leftmost *len* characters of the constant are transmitted to the list item. If *len* is greater than *w*, the constant is transmitted to the leftmost *w* characters of the list item and the remaining *len* – *w* characters of the list item are filled with blanks. Note that the effect is as though the constant were assigned to the list item in a character assignment statement (7.5.1.4).

10.9.1.4 Null Values. A null value is specified by:

(1) The *r*∗ form,

(2) Blanks between two consecutive value separators following an equals,

(3) Zero or more blanks preceding the first value separator and following an equals, or

(4) Two consecutive nonblank value separators.

A null value has no effect on the definition status of the corresponding input list item. If the namelist group object list item is defined, it retains its previous value; if it is undefined, it remains undefined. A null value must not be used as either the real or imaginary part of a complex constant, but a single null value may represent an entire complex constant.

Note that the end of a record following a value separator, with or without intervening blanks, does not specify a null value.

10.9.1.5 Blanks. All blanks in a namelist input record are considered to be part of some value separator except for:

(1) Blanks embedded in a character constant,

(2) Embedded blanks surrounding the real or imaginary part of a complex constant,

(3) Leading blanks following the equals unless followed immediately by a slash or comma, and

(4) Blanks between a name and the following equals.

10.9.2 Namelist Output. The form of the output produced is the same as that required for input, except for the forms of real and logical constants. If the processor is capable of representing letters in both upper and lower case, the name in the output is in upper case. With the exception of nondelimited character constants, the values are separated by (1) one or more blanks or (2) a comma ~~optionally~~ preceded by ~~one~~ zero or more blanks and ~~optionally~~ followed by ~~one~~ zero or more blanks.

The processor may begin new records as necessary. However, except for complex constants and character constants, the end of a record must not occur within a constant or a name, and

blanks must not appear within a constant or a name.

10.9.2.1 Namelist Output Editing. Logical output constants are T for the value true and F for the value false.

Integer output constants are produced with the effect of an Iw edit descriptor.

Real constants are produced with the effect of either an F edit descriptor or an E edit descriptor, depending on the magnitude x of the value and a range $10^{d_1} \le x < 10^{d_2}$, where d_1 and d_2 are processor-dependent integers. If the magnitude x is within this range, the constant is produced using 0PF*w.d*; otherwise, 1PE*w.d*E*e* is used.

For numeric output, reasonable processor-dependent integer values of *w*, *d*, and *e* are used for each of the cases involved.

Complex constants are enclosed in parentheses with a comma separating the real and imaginary parts. The end of a record may occur between the comma and the imaginary part only if the entire constant is as long as, or longer than, an entire record. The only embedded blanks permitted within a complex constant are between the comma and the end of a record and one blank at the beginning of the next record.

Character constants produced for a file opened without a DELIM= specifier (9.3.4.9) or with a DELIM= specifier with a value of NONE:

(1) Are not delimited by apostrophes or quotation marks,

(2) Are not preceded or followed by a value separator,

(3) Have each internal apostrophe or quotation mark represented externally by one apostrophe or quotation mark, and

(4) Have a blank character inserted by the processor for carriage control at the beginning of any record that begins with the continuation of a character constant from the preceding record.

Character constants produced for a file opened with a DELIM= specifier with a value of QUOTE are delimited by quotes, are preceded and followed by a value separator, and have each internal quote represented on the external medium by two contiguous quotes.

Character constants produced for a file opened with a DELIM= specifier with a value of APOSTROPHE are delimited by apostrophes, are preceded and followed by a value separator, and have each internal apostrophe represented on the external medium by two contiguous apostrophes.

10.9.2.2 Namelist Output Records. If two or more successive values in an array in an output record produced have identical values, the processor has the option of producing a repeated constant of the form *r***c* instead of the sequence of identical values.

The name of each namelist group object list item is placed in the output record followed by an equals and one or more values of the namelist group object list item.

An ampersand character followed immediately by a *namelist-group-name* will be produced by namelist formatting at the start of the first output record to indicate which specific group of data objects is being output. A slash is produced by namelist formatting to indicate the end of the namelist formatting.

A null value is not produced by namelist formatting.

Except for continuation of delimited character constants, each output record begins with a blank character to provide carriage control when the record is printed.

11. PROGRAM UNITS

The terms and basic concepts of program units were introduced in 2.2. A program unit may be a main program, an external subprogram, a module, or a block data program unit.

This section describes all of these program units except external subprograms, which are described in Section 12.

11.1 Main Program.

R1101 *main-program* **is** [*program-stmt*]
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-program-stmt

R1102 *program-stmt* **is** PROGRAM *program-name*

R1103 *end-program-stmt* **is** END [PROGRAM [*program-name*]]

Constraint: In a *main-program*, the *execution-part* must not contain a RETURN statement or an ENTRY statement.

Constraint: The *program-name* may be included in the *end-program-stmt* only if the optional *program-stmt* is used and, if included, must be identical to the *program-name* specified in the *program-stmt*.

The **program name** is global to the executable program, and must not be the same as the name of any other program unit, external procedure, or common block in the executable program, nor the same as any local name in the main program. entry point

An example of a main program is:

```
PROGRAM ANALYSE
   REAL A, B, C (10,10)      !  SPECIFICATION PART
   CALL FIND                 !  EXECUTION PART
CONTAINS
   SUBROUTINE FIND           !  INTERNAL PROCEDURE
      . . .
   END SUBROUTINE FIND
END PROGRAM ANALYSE
```

11.1.1 Main Program Specifications. The specifications in the scoping unit of the main program must not include an OPTIONAL statement, an INTENT statement, a PUBLIC statement, a PRIVATE statement, or the equivalent attributes (5.1.2). A SAVE statement has no effect in a main program. The specification part must not declare an automatic object.

11.1.2 Main Program Executable Part. The sequence of *execution-part* statements specifies the actions of the main program during program execution. Execution of an executable program (R201) begins with the first executable construct of the main program.

A main program ~~must~~ can not be recursive; that is, a reference to it ~~must~~ can not appear in any program unit in the executable program, including itself.

Execution of an executable program ends with execution of the *end-program-stmt* of the main program or with execution of a STOP statement in any program unit of the executable program.

11.1.3 Main Program Internal Procedures. Any definitions of internal procedures in the main program must follow the CONTAINS statement. Internal procedures are described in 11.2.1. The main program is called the **host** of its internal procedures.

11.2 Procedures. External procedures and module procedures are described in Section 12.

11.2.1 Internal Procedures. Internal procedures may appear in the main program, in an external subprogram, or in a module subprogram. Internal procedures are the same as external procedures except that the name of the internal procedure is not a global entity, an internal procedure must not contain an ENTRY statement, the internal procedure name must not be argument associated with a dummy procedure (12.4.1.2), and the internal procedure has access to host entities by host association.

11.2.2 Host Association. An internal subprogram, a module subprogram, or a derived-type definition has access to the named entities from its host via **host association**. The accessed entities are known by the same name and have the same attributes as in the host and include variables, constants, procedures including interfaces, derived types, type parameters, derived-type components, and namelist groups.

Any declaration or specification of an entity with the same name as an accessible entity from the host makes the host entity inaccessible. The appearance of a name as a dummy argument or as a function result is treated as a specification.

Note that an interface block does not access the named entities from its host by host association.

If a procedure gains access to a pointer by host association, the association of the pointer with a target that is current at the time the procedure is invoked remains current within the procedure. This pointer association may be changed within the procedure by allocation, deallocation, or pointer assignment. When execution of the procedure completes, the pointer association that was current remains current, except where the associated target was declared within the procedure and is not saved. In this case, the completion of the procedure causes the pointer association status of the host associated pointer to become undefined. Such a pointer may not be used in any way until its association status is reestablished.

11.3 Modules. A module contains specifications and definitions that are to be accessible to other program units.

R1104 *module* **is** *module-stmt*
[*specification-part*]
[*module-subprogram-part*]
end-module-stmt

R1105 *module-stmt* **is** MODULE *module-name*

R1106 *end-module-stmt* **is** END [MODULE [*module-name*]]

Constraint: If the *module-name* is specified in the *end-module-stmt*, it must be identical to the *module-name* specified in the *module-stmt*.

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

Constraint: An automatic object must not appear in the *specification-part* (R203) of a module.

The module name is global to the executable program, and must not be the same as the name of any other program unit, external procedure, or common block in the executable program, nor the same as any local name in the module.

11.3.1 Module Reference. A USE statement specifying a module name is a **module reference**. At the time a module reference is processed, the public portions of the specified module must be available. A module must not reference itself, either directly or indirectly.

The accessibility, public or private, of specifications and definitions in a module to a scoping unit making reference to the module may be controlled in both the module and the scoping unit making the reference. In the module, the PRIVATE statement, the PUBLIC statement (5.2.3), the

equivalent attributes (5.1.2.2), and the PRIVATE statement in a derived-type definition (4.4.1) are used to control the accessibility of module entities outside the module.

In a scoping unit making reference to a module, the ONLY option on the USE statement may be used to further limit the accessibility, in that referencing scoping unit, of the public entities in the module.

11.3.2 The USE Statement. The **USE statement** provides the means by which a scoping unit accesses named data objects, derived types, interface blocks, procedures, and namelist groups in a module. The entities in the scoping unit are said to be **use associated** with the entities in the module. The accessed entities have the attributes specified in the module.

R1107 *use-stmt* **is** USE *module-name* [, *rename-list*]
or USE *module-name* , ONLY : [*only-list*]

R1108 *rename* **is** *local-name* => *use-name*

R1109 *only* **is** [*local-name*] => *use-name*

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

Each *use-name* must be the name of a public entity in the module. If a *local-name* appears in a *rename-list* or an *only-list*, it is the local name for the entity specified by *use-name*; otherwise, the local name is the *use-name*.

The USE statement without the ONLY option provides access to all public entities in the specified module.

A USE statement with the ONLY option provides access only to those entities whose names appear as *use-names* in the *only-list*. In a scoping unit, two or more accessible entities may have the same name only if no entity is referenced by this name in the scoping unit. Except for this, the local name of any entity given accessibility by a USE statement must differ from the local names of all other entities accessible to the scoping unit through USE statements and otherwise. Note that an entity may be accessed by more than one local name.

The local name of an entity made accessible by a USE statement may appear in no other specification statement that would cause any attribute of the entity to be respecified in the scoping unit that contains the USE statement, except that it may appear in a PUBLIC or PRIVATE statement in the scoping unit of a module. The appearance of such a local name in a PUBLIC statement in a module causes the entity accessible by the USE statement to be a public entity of that module. If the name appears in a PRIVATE statement in a module, the entity is not a public entity of that module. If the local name does not appear in either a PUBLIC or PRIVATE statement, it assumes the default accessibility attribute (5.2.3) of that scoping unit.

Note that the above rule prohibits the local name from appearing in COMMON and EQUIVALENCE specifications, but permits the appearance of local names in NAMELIST group lists.

Examples:

```
USE STATS_LIB
```

provides access to all public entities in the module STATS_LIB.

```
USE MATH_LIB; USE STATS_LIB, SPROD => PROD
```

makes all public entities in both MATH_LIB and STATS_LIB accessible. If MATH_LIB contains an entity called PROD, it is accessible by its own name while the entity PROD of STATS_LIB is accessible by the name SPROD. Both modules may contain an entity called SUMM, for example, if SUMM does not appear in the scoping unit containing the USE statements and SUMM is not declared in a type statement in the scoping unit.

11.3.3 Examples of the Use of Modules.

11.3.3.1 Identical Common Blocks. A common block and all its associated specification statements may be placed in a module named, for example, MY_COMMON and accessed by a USE statement of the form

```
USE MY_COMMON
```

that accesses the whole module without any renaming. This ensures that all instances of the common block are identical. Module MY_COMMON could contain more than one common block.

11.3.3.2 Global Data. A module may contain only data objects, for example:

```
MODULE DATA_MODULE
   REAL A (10), B, C (20,20)
   INTEGER :: I=0
   INTEGER, PARAMETER :: J=10
   COMPLEX D (J,J)
END MODULE
```

Note that data objects made global in this manner may have any combination of data types.

Access to some of these may be made by a USE statement with the ONLY option, such as:

```
USE DATA_MODULE, ONLY: A, B, D
```

and access to all of them may be made by the following USE statement:

```
USE DATA_MODULE
```

Access to all of them with some renaming to avoid name conflicts may be made by:

```
USE DATA_MODULE, AMODULE => A, DMODULE => D
```

11.3.3.3 Data Structures. A derived type may be defined in a module and accessed in a number of program units. This is the only way to access the same type definition in more than one program unit. For example:

```
MODULE SPARSE
   TYPE NONZERO
      REAL A
      INTEGER  I, J
   END TYPE
END MODULE
```

defines a type consisting of a real component and two integer components for holding the numerical value of a nonzero matrix element and its row and column indices.

11.3.3.4 Global Allocatable Arrays. Many programs need large global allocatable arrays whose sizes are not known before program execution. A simple form for such a program is:

```
PROGRAM GLOBAL_WORK
   CALL CONFIGURE_ARRAYS       ! PERFORM THE APPROPRIATE ALLOCATIONS
   CALL COMPUTE                ! USE THE ARRAYS IN COMPUTATIONS
END PROGRAM GLOBAL_WORK

MODULE WORK_ARRAYS             ! AN EXAMPLE SET OF WORK ARRAYS
   INTEGER  N
   REAL, ALLOCATABLE, SAVE :: A(:), B(:, :), C(:, :, :)
END MODULE WORK_ARRAYS

SUBROUTINE CONFIGURE_ARRAYS    ! PROCESS TO SET UP WORK ARRAYS
   USE WORK_ARRAYS
   READ (INPUT, *)  N
```

```
   ALLOCATE (A (N), B (N, N), C (N, N, 2 * N))
END SUBROUTINE CONFIGURE_ARRAYS

SUBROUTINE COMPUTE
   USE WORK_ARRAYS
   ... ! COMPUTATIONS INVOLVING ARRAYS A, B, AND C
END SUBROUTINE COMPUTE
```

Typically, many subprograms need access to the work arrays, and all such subprograms would contain the statement

```
USE WORK_ARRAYS
```

11.3.3.5 Procedure Libraries. Interfaces to external procedures in a library may be gathered into a module. This permits the use of argument keywords and optional arguments, and allows static checking of the references. Different versions may be constructed for different applications, using keywords in common use in each application. An example is the following library module:

```
MODULE LIBRARY_LLS
   INTERFACE
      SUBROUTINE LLS (X, A, F, FLAG)
      REAL (KIND (0.0)) X (:, :)
      REAL (KIND (0.0)), DIMENSION (SIZE (X, 2)) :: A, F
      INTEGER FLAG
   END INTERFACE
END MODULE
```

This module allows the subroutine LLS to be invoked:

```
USE LIBRARY_LLS
   ...
CALL LLS (X = ABC, A = D, F = XX, FLAG = IFLAG)
   ...
```

11.3.3.6 Operator Extensions. To extend an intrinsic operator symbol to have an additional meaning, an interface block specifying that operator symbol in the OPERATOR option of the INTERFACE statement may be placed in a module. For example, // may be overloaded to perform concatenation of two derived-type objects serving as varying length character strings and + may be overloaded to specify matrix addition or interval arithmetic addition.

A module might contain several such interface blocks. If the operation is written in a language other than Fortran, it may be written as an external function and its procedure interface placed in the module.

11.3.3.7 Data Abstraction. A module may encapsulate a derived-type definition and all the procedures that represent operations on values of this type. An example is given in Appendix C for set operations.

11.4 Block Data Program Units.

A block data program unit is used to provide initial values for data objects in ~~named~~ common blocks.

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

R1111 *block-data-stmt* **is** BLOCK DATA [*block-data-name*]

R1112 *end-block-data-stmt* **is** END [BLOCK DATA [*block-data-name*]]

Constraint: The *block-data-name* may be included in the *end-block-data-stmt* only if it was provided in the *block-data-stmt* and, if included, must be identical to the *block-data-name* in the *block-data-stmt*.

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

If an object in a ~~named~~ common block is initially defined, all objects having storage units in the common block storage sequence must be specified even if they are not all initially defined. More than one ~~named~~ common block may have objects initially defined in a single block data program unit. Note, therefore, that the primary constituents of a block data program unit are type declarations of common block objects, COMMON statements, and DATA statements.

Only an object in a ~~named~~ common block may be initially defined in a block data program unit. Note that objects associated with an object in a common block are considered to be in that common block.

The same ~~named~~ common block must not be specified in more than one block data program unit in an executable program.

There must not be more than one unnamed block data program unit in an executable program.

An example of a block data program unit is:

```
BLOCK DATA WORK
   COMMON /WRKCOM/ A, B, C (10, 10)
   DATA A /1.0/, B /2.0/, C /100 * 0.0/
END BLOCK DATA WORK
```

12. PROCEDURES

The concept of a procedure was introduced in 2.2.4. This section contains a complete description of procedures. The action specified by a procedure is performed when the procedure is invoked by execution of a reference to it. The reference may identify, as actual arguments, entities that are associated during execution of the procedure reference with corresponding dummy arguments in the procedure definition.

12.1 Procedure Classifications. A procedure is classified according to the form of its reference and the way it is defined.

12.1.1 Procedure Classification by Reference. The definition of a procedure specifies it to be a function or a subroutine. A reference to a function either appears explicitly as a primary within an expression, or is implied by a defined operation within an expression. A reference to a subroutine is a CALL statement or a defined assignment statement (7.5.1.3).

A procedure is classified as **elemental** if it may be referenced elementally (12.4.3).

12.1.2 Procedure Classification by Means of Definition. A procedure is either an intrinsic procedure, an external procedure, a module procedure, an internal procedure, a dummy procedure, or a statement function.

12.1.2.1 Intrinsic Procedures. A procedure that is provided as an inherent part of the processor is an **intrinsic procedure**.

12.1.2.2 External, Internal, and Module Procedures. An **external procedure** is a procedure that is defined by an external subprogram or by a means other than Fortran.

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

A **module procedure** is a procedure that is defined by a module subprogram.

If a subprogram contains one or more ENTRY statements, it defines a procedure for each ENTRY statement and a procedure for the SUBROUTINE or FUNCTION statement.

An internal subprogram must not contain an ENTRY statement.

12.1.2.3 Dummy Procedures. A dummy argument that is specified as a procedure or appears in a procedure reference is a **dummy procedure**.

12.1.2.4 Statement Functions. A function that is defined by a single statement is a **statement function** (12.5.4).

12.2 Characteristics of Procedures. The **characteristics of a procedure** are the classification of the procedure as a function or subroutine, the characteristics of its arguments, and the characteristics of its result value if it is a function.

12.2.1 Characteristics of Dummy Arguments. Each dummy argument is either a dummy data object, a dummy procedure, or an asterisk (alternate return indicator). A dummy argument other than an asterisk may be specified to have the OPTIONAL attribute. This attribute means that the dummy argument need not be associated with an actual argument for any particular reference to the procedure.

12.2.1.1 Characteristics of Dummy Data Objects. The characteristics of a dummy data object are its type, type parameters (if any), shape, intent (5.1.2.3, 5.2.1), optionality (5.1.2.6, 5.2.2), and whether it is a pointer (5.1.2.7) or a target (5.1.2.8). If a type parameter of an object or a bound of an array is an expression that depends on the value or attributes of another object, the exact dependence on other entities is a characteristic. If a shape, size, or character length is assumed, it is a characteristic.

12.2.1.2 Characteristics of Dummy Procedures. The characteristics of a dummy procedure are the explicitness of its interface (12.3.1), the characteristics of the procedure if the interface is explicit, and its optionality (5.1.2.6, 5.2.2).

12.2.1.3 Characteristics of Asterisk Dummy Arguments. An asterisk as a dummy argument has no characteristics.

12.2.2 Characteristics of Function Results. The characteristics of a function result are its type, type parameters (if any), rank, and whether it is a pointer. If a function result is an array that is not allocatable, its shape is a characteristic. Where a type parameter or bound of an array is not a constant expression, the exact dependence on the entities in the expression is a characteristic. If the length of a character data object is assumed, this is a characteristic.

12.3 Procedure Interface.

The **interface** of a procedure determines the forms of reference through which it may be invoked. The interface consists of the characteristics of the procedure, the name of the procedure, the name and characteristics of each dummy argument, the defined operator (if any) by which a reference to a function may appear, and whether or not a reference to a subroutine may be implied in a defined assignment statement. The characteristics of a procedure are fixed, but the remainder of the interface may differ in different scoping units.

12.3.1 Implicit and Explicit Interfaces. If a procedure is accessible in a scoping unit, its interface is either **explicit** or **implicit** in that scoping unit. The interface of an internal procedure, module procedure, or intrinsic procedure is always explicit in such a scoping unit. The interface of a statement function is always implicit. The interface of an external procedure or dummy procedure is explicit if an interface block (12.3.2.1) for the procedure is supplied or accessible, and implicit otherwise. For example, the subroutine LLS of 11.3.3.5 has an explicit interface.

12.3.1.1 Explicit Interface. A procedure must have an explicit interface if any of the following is true:

(1) A reference to the procedure appears:
   (a) With an argument keyword (12.4.1)
   (b) As a defined assignment (subroutines only)
   (c) In an expression as a defined operator (functions only)

(2) The procedure has:
   (a) An optional dummy argument
   (b) An array-valued result (functions only)
   (c) A dummy argument that is an assumed-shape array, a pointer, or a target.
   (d) A result whose type parameter values are neither assumed length (character type only) nor constant (functions only)

(3) Another procedure having the same generic name is accessible (12.3.2.1)

12.3.2 Specification of the Procedure Interface. The interface for an internal, external, module, or dummy procedure is specified by a FUNCTION, SUBROUTINE, or ENTRY statement and by specification statements for the dummy arguments and the result of a function. These statements may appear in the procedure definition, in an interface block, or both except that the ENTRY statement must not appear in an interface block.

12.3.2.1 Procedure Interface Block.

R1201 *interface-block* **is** *interface-stmt*
[*procedure-interface*] ...
[*module-procedure-stmt*] ...

end-interface-stmt

R1202 *interface-stmt* **is** INTERFACE [*generic-spec*]

R1203 *end-interface-stmt* **is** END INTERFACE

R1204 *procedure-interface* **is** *procedure-heading*
[*use-stmt*] ...
[*implicit-part*]
[*declaration-construct*] ...
procedure-ending

R1205 *module-procedure-stmt* **is** MODULE PROCEDURE *procedure-name-list*

R1206 *generic-spec* **is** *generic-name*
or OPERATOR (*defined-operator*)
or ASSIGNMENT (=)

Constraint: An *interface-block* must not contain an *entry-stmt*.

Constraint: The MODULE PROCEDURE specification is allowed only if the *interface-block* has a *generic-spec*.

Constraint: An interface block must not appear in a BLOCK DATA program unit.

An external or module subprogram definition specifies a **specific interface** for the procedures defined in that subprogram. Such a specific interface is explicit for module procedures and implicit for external procedures. A **procedure interface** in an interface block specifies an explicit interface for an existing external procedure or a dummy procedure.

An external procedure interface specifies all of the procedure's characteristics and these must be consistent with those specified in the procedure definition, except that the dummy argument names may be different. An interface block must not contain an ENTRY statement, but an ENTRY interface may be specified by using the entry name as the procedure name in the interface block. A procedure must not have more than one explicit specific interface in a given scoping unit.

An example of an interface block without a generic specification is:

```
INTERFACE

   SUBROUTINE EXT1 (X, Y, Z)
      REAL, DIMENSION (100, 100) :: X, Y, Z
   END SUBROUTINE EXT1

   SUBROUTINE EXT2 (X, Z)
      COMPLEX (KIND = 4) Z (2000)
   END SUBROUTINE EXT2

   FUNCTION EXT3 (P, Q)
      LOGICAL EXT3
      INTEGER P (1000)
      LOGICAL Q (1000)
   END FUNCTION EXT3

END INTERFACE
```

This interface block specifies explicit interfaces for the three external procedures EXT1, EXT2, and EXT3. Any of these procedures may use keyword calls; for example:

```
EXT3 (Q = P_MASK (N+1 : N+1000), P = ACTUAL_P)
```

An interface block with a generic specification specifies a **generic interface** for each of the procedures in the interface block. The MODULE PROCEDURE specification lists those module procedures, either defined in that module (if the interface block is in a module) or accessible via a USE

statement, that have this generic interface. The characteristics of module procedures are not given in interface blocks, but are assumed from the module subprogram definitions. A procedure must not have more than one generic interface in a given scoping unit. A generic interface is always explicit.

A procedure always may be referenced via its specific interface. It also may be referenced via its generic interface, if it has one.

A generic name **overloads** all of the procedure names in the interface block. A generic name may be the same as any one of the procedure names in that interface block, or the same as the name of any accessible module procedure. Any overload is allowed for which any given procedure interface would apply to at most one procedure. The specific rules on how any two such procedure interfaces must differ are given in 14.1.2.3.

An example of a generic procedure interface is:

```
INTERFACE SWITCH

   SUBROUTINE INT_SWITCH (X, Y)
      INTEGER, INTENT (INOUT) :: X, Y
   END SUBROUTINE INT_SWITCH

   SUBROUTINE REAL_SWITCH (X, Y)
      REAL, INTENT (INOUT) :: X, Y
   END SUBROUTINE REAL_SWITCH

   SUBROUTINE COMPLEX_SWITCH (X, Y)
      COMPLEX, INTENT (INOUT) :: X, Y
   END SUBROUTINE COMPLEX_SWITCH

END INTERFACE
```

Any of these three subroutines (INT_SWITCH, REAL_SWITCH, COMPLEX_SWITCH) may be referenced with the generic name SWITCH, as well as by its original name. For example, a reference to INT_SWITCH could take the form:

```
CALL SWITCH (MAX_VAL, LOC_VAL) ! MAX_VAL and LOC_VAL are of type INTEGER
```

If OPERATOR is specified in a generic specification, all of the procedures specified in the interface block must be functions, which may be referenced alternatively as defined operations (12.4). In the case of functions of two arguments, infix binary operator notation is implied. In the case of one-argument functions, prefix operator notation is implied. OPERATOR must not be specified for functions with no arguments or for functions with more than two arguments. In addition, the dummy arguments must be nonoptional dummy data objects and may be specified with INTENT (IN) but not INTENT (OUT) or INTENT (INOUT); INTENT (IN) is assumed if intent is not specified.

A defined operation is treated as a reference to the function. For a unary defined operation, the operand corresponds to the function's dummy argument; for a binary operation, the left-hand operand corresponds to the first dummy argument of the function and the right hand operand corresponds to the second argument.

An example of the use of the OPERATOR generic specification is:

```
INTERFACE OPERATOR ( * )

   FUNCTION BOOLEAN_AND (B1, B2)
      LOGICAL, INTENT (IN) :: BOOLEAN_AND (SIZE (B1))
      LOGICAL, INTENT (IN) :: B1 (:), B2 (SIZE (B1))
   END FUNCTION BOOLEAN_AND

END INTERFACE
```

This allows, for example

```
SENSOR (1:N) * ACTION (1:N)
```

as an alternative to the function call

```
BOOLEAN_AND (SENSOR (1:N), ACTION (1:N))    ! SENSOR and ACTION are
                                            ! of type LOGICAL
```

A given defined operator may, as with generic names, apply to more than one function, in which case it is overloaded in exact analogy to overloaded procedure names. For intrinsic operator symbols, the overloads include the intrinsic operations they represent. The procedure characteristic rules for generic name overloading apply without change to operator overloading.

If ASSIGNMENT is specified in an INTERFACE statement, all the procedures in the interface block must be subroutines, which alternatively may be referenced as defined assignments (12.3.2.1). Each of these subroutines must have exactly two dummy arguments. The first argument may have INTENT (OUT) or INTENT (INOUT) and the second argument may have INTENT (IN); for unspecified intent, INTENT (OUT) is assumed for the first argument and INTENT (IN) is assumed for the second argument. A defined assignment is treated as a reference to the subroutine, with the left-hand side as the first argument and the right-hand side enclosed in parentheses as the second argument. The ASSIGNMENT generic specification specifies that the assignment operation is overloaded and, with the inclusion of intrinsic assignment, the procedure characteristics rules for generic name overloading apply to assignment overloading.

An example of the use of the ASSIGNMENT generic specification is

```
INTERFACE ASSIGNMENT ( = )

   SUBROUTINE BIT_TO_NUMERIC (N, B)
      INTEGER, INTENT (OUT) :: N
      LOGICAL, INTENT (IN)  :: B(:)
   END SUBROUTINE BIT_TO_LOGICAL

   SUBROUTINE CHAR_TO_STRING (S, C)
      TYPE (STRING), INTENT (OUT) :: S  ! A variable-length string
      CHARACTER (*), INTENT (IN)  C
   END SUBROUTINE CHAR_TO_STRING

END INTERFACE
```

Example assignments are:

```
KOUNT = SENSOR (J:K)    ! CALL BIT_TO_NUMERIC (KOUNT, (SENSOR (J:K)))
NOTE  = '89AB'          ! CALL CHAR_TO_STRING (NOTE, ('89AB'))
```

12.3.2.2 EXTERNAL Statement. An **EXTERNAL statement** is used to specify a name as representing an external procedure or dummy procedure, and to permit such a name to be used as an actual argument.

R1207 *external-stmt* **is** EXTERNAL *external-name-list*

Each *external-name* must be the name of an external procedure, a dummy argument, or a block data program unit.

The appearance of the name of a dummy argument in an EXTERNAL statement specifies that the dummy argument is a dummy procedure.

The appearance in an EXTERNAL statement of a name that is not the name of a dummy argument specifies that the name is the name of an external procedure or block data program unit. If an external procedure name or a dummy procedure name is used as an actual argument, it must either appear in an EXTERNAL 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.

Only one appearance of a name in all of the EXTERNAL statements in a scoping unit is permitted.

An example of an EXTERNAL statement is:

```
SUBROUTINE SUB (FOCUS)
   EXTERNAL FOCUS
```

12.3.2.3 INTRINSIC Statement. An **INTRINSIC statement** is used to specify a name as representing an intrinsic procedure (Section 13). It also permits a name that represents a specific intrinsic function to be used as an actual argument.

R1208 *intrinsic-stmt* **is** INTRINSIC *intrinsic-procedure-name-list*

The appearance of a name in an INTRINSIC statement confirms that the name is the name of an intrinsic procedure. The appearance of a generic function name (13.10) in an INTRINSIC statement does not cause that name to lose its generic property.

If the specific name (13.12) of an intrinsic function is used as an actual argument, the name must appear in an INTRINSIC statement in the scoping unit.

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.

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.

12.4 Procedure Reference. The form of a procedure reference is dependent on the interface of the procedure, but is independent of the means by which the procedure is defined. The forms of procedure references are:

R1209 *function-reference* **is** *function-name* ([*actual-arg-spec-list*])
or *defined-operation*

why not?

Constraint: The *actual-arg-spec-list* for a function reference must not contain an *alt-return-spec*.

R1210 *defined-operation* **is** [*actual-arg*] *defined-operator actual-arg*

R1211 *subroutine-reference* **is** *call-stmt*
or *defined-assignment*

R1212 *call-stmt* **is** CALL *subroutine-name* [([*actual-arg-spec-list*])]

R1213 *defined-assignment* **is** *actual-arg* = *actual-arg*

Constraint: *actual-arg* for a *defined-operation* or *defined-assignment* must not be a procedure name or *alt-return-spec*.

12.4.1 Actual Argument List.

R1214 *actual-arg-spec* **is** [*keyword* =] *actual-arg*

R1215 *keyword* **is** *dummy-arg-name*

R1216 *actual-arg* **is** *expr*
or *variable*
or *procedure-name*
or alt-return-spec

R1217 *alt-return-spec* is * label

Constraint: The *keyword* may be omitted from an *actual-arg-spec* only if the *keyword* has been omitted from each preceding *actual-arg-spec* in the argument list.

Constraint: Each *keyword* must be the name of a dummy argument in the explicit interface of the procedure.

Constraint: A *procedure-name actual-arg* must not be the name of an internal procedure and must not be the name of an intrinsic subroutine (13.9). If it is the name of an intrinsic function, it must be a specific name for the function (13.12).

why?

Constraint: The *label* used in the *alt-return-spec* must be the statement label of a branch target statement that appears in the same scoping unit as the *call-stmt*.

In either a subroutine reference or a function reference, the actual argument list identifies the correspondence between the actual arguments supplied and the dummy arguments of the procedure. In the absence of a keyword, an actual argument is associated with the dummy argument occupying the corresponding position in the dummy argument list; i.e., the first actual argument is associated with the first dummy argument, the second actual argument is associated with the second dummy argument, etc. If a keyword is present, the actual argument is associated with the dummy argument whose name is the same as the keyword. Exactly one actual argument must be associated with each nonoptional dummy argument. At most one actual argument may be associated with each optional dummy argument. Each actual argument must be associated with a dummy argument. For example, the procedure

```
SUBROUTINE SOLVE (FUNCT, SOLUTION, METHOD, STRATEGY, PRINT)
   INTERFACE
      FUNCTION FUNCT (X)
      REAL FUNCT, X
   END INTERFACE
   REAL SOLUTION
   INTEGER, OPTIONAL :: METHOD, STRATEGY, PRINT
   ...
```

may be invoked with

```
CALL SOLVE (FUN, SOL, PRINT = 6)
```

providing its interface is explicit.

12.4.1.1 Arguments Associated with Dummy Data Objects. If a dummy argument is a dummy data object, the associated actual argument must be an expression of the same type or a data object of the same type. The type parameter values of the actual argument, if any, must agree with or be assumed by the dummy argument, except that a character actual argument may have a length greater than that of the character dummy argument, in which case the effect is as if the leftmost substring of appropriate length were used as the actual argument. Except when a procedure reference is elemental (12.4.3), each element of an array-valued actual argument or of a sequence in a sequence association (12.4.1.4) is associated with the element of the dummy array that has the same position in array element order (6.2.2.2).

If the dummy argument is a pointer, the actual argument must be a pointer and the types, type parameters, and ranks must agree. The dummy argument pointer becomes associated with the target of the actual argument pointer at the invocation of the procedure. This association may be changed during the execution of the procedure, either by allocation or by pointer assignment. When execution of the procedure completes, the association of the actual argument pointer with a target becomes that of the dummy argument, except where the dummy argument target was declared within the procedure and is not saved. In this case, the association status of the actual argument pointer with a target is undefined. Such a pointer may not be used in any way until its association status is reestablished by the execution of an ALLOCATE, pointer assignment, or NULLIFY statement.

If the actual argument is scalar, the corresponding dummy argument must be scalar unless the actual argument is an element of a named array that is explicit-shaped or assumed-size, or a substring of such an element. If the procedure is referenced by a generic name or as a defined operator or defined assignment, the ranks of the actual arguments and corresponding dummy arguments must agree.

If a dummy argument has INTENT (OUT) or INTENT (INOUT), the actual argument must be definable. If a dummy argument has INTENT (OUT), the corresponding actual argument becomes undefined at the time the association is established.

If the actual argument is an array section having a vector subscript, the dummy argument is not definable and must not have INTENT (OUT) or INTENT (INOUT).

12.4.1.2 Arguments Associated with Dummy Procedures. If a dummy argument is a dummy procedure, the associated actual argument must be the name of an external, module, dummy, or intrinsic procedure. The only intrinsic procedures permitted are those listed in 13.12 and not marked with a bullet (•). The actual argument name must be one for which exactly one procedure is accessible in the invoking scoping unit. (A specific intrinsic function and a generic intrinsic function of the same name are considered to be one procedure.)

If the interface of the dummy procedure is explicit, the characteristics of the associated procedure must be the same as the characteristics of the dummy procedure (12.2).

If the interface of the dummy procedure is implicit and either the name of the dummy procedure is explicitly typed or the procedure is referenced as a function, the dummy procedure must not be referenced as a subroutine and the actual argument must be a function or dummy procedure.

If the interface of the dummy procedure is implicit and a reference to the procedure appears as a subroutine reference, the actual argument must be a subroutine or dummy procedure.

12.4.1.3 Arguments Associated with Alternate Return Indicators. If a dummy argument is an asterisk (12.5.2.3), the associated actual argument must be an alternate return specifier. The label in the alternate return specifier must identify an executable construct in the scoping unit containing the procedure reference.

12.4.1.4 Sequence Association. An actual argument represents an **element sequence** if it is an array name, an array element designator, or an array element substring designator. If the actual argument is an array name, the element sequence consists of the elements in array element order. If the actual argument is an array element designator, the element sequence consists of that array element and each element that follows it in array element order. If the actual argument is an array element substring designator, the element sequence consists of the character storage units beginning with the first storage unit in that array element substring and continuing to the end of the array. The character storage units are viewed as elements consisting of consecutive groups of character storage units having the length of the array element substring. Thus, the first such element is the array element substring itself. Note that some of the elements in the element sequence may consist of storage units from different elements of the original array.

An actual argument that represents an element sequence and corresponds to a dummy argument that is an array-valued data object is sequence associated with the dummy argument if the interface is implicit or if the dummy argument is an explicit-shape or assumed-size array. The rank and shape of the actual argument need not agree with the rank and shape of the dummy argument, but the number of elements in the dummy argument must not exceed the number of elements in the element sequence of the actual argument. If the dummy argument is assumed size, the number of elements in the dummy argument is exactly the number of elements in the element sequence.

12.4.2 Function Reference. A function is invoked during expression evaluation by a function reference or by a defined operation (7.1.3). When it is invoked, all actual argument expressions are evaluated, then the arguments are associated, and then the function is executed. When execution of the function is complete, the value of the function result is available for use in the

expression that caused the function to be invoked. The characteristics of the function result (12.2.2) are determined by the interface of the function.

12.4.3 Elemental Intrinsic Function Reference. A reference to an elemental intrinsic function is an **elemental reference** if one or more actual arguments are arrays that have the same shape. The result has the same shape as the array arguments and the value of each element in the result is obtained by evaluating the function using the scalar arguments and the corresponding elements of the array arguments. For example, if X and Y are arrays of shape (*m, n*),

```
MAX (X, 0.0, Y)
```

is an array expression of shape (*m, n*) whose elements have values

MAX (X (*i, j*), 0.0, Y (*i, j*)), *i* = 1, 2, ..., *m*, *j* = 1, 2, ..., *n*

12.4.4 Subroutine Reference. A subroutine is invoked by execution of a CALL statement or defined assignment statement (7.5.1.3). When a subroutine is invoked, all actual argument expressions are evaluated, then the arguments are associated, and then the subroutine is executed. When the action specified by the subroutine is completed, execution of the CALL statement or defined assignment statement is also completed. If a CALL statement includes one or more alternate return specifiers among its arguments, control may be transferred to one of the statements indicated, depending on the action specified by the subroutine.

12.4.5 Elemental Subroutine Reference. A reference to an elemental intrinsic subroutine is an elemental reference if all output actual arguments are arrays that have the same shape and the remaining actual arguments are conformable with them. The values of the elements of the output arrays are the same as would be obtained if the subroutine were applied separately to corresponding elements of each argument.

12.5 Procedure Definition.

12.5.1 Intrinsic Procedure Definition. Intrinsic procedures are defined as an inherent part of the processor. A standard-conforming processor must include the intrinsic procedures described in Section 13, but may include others. However, a standard-conforming program must not make use of intrinsic procedures other than those described in Section 13.

12.5.2 Procedures Defined by Subprograms. When a procedure defined by a subprogram is invoked, an instance (12.5.2.4) of the procedure is created and executed. Execution begins with the first executable construct following the FUNCTION, SUBROUTINE, or ENTRY statement specifying the name of the procedure invoked.

12.5.2.1 Effects of INTENT Attribute on Subprograms. The INTENT attribute of dummy data objects limits the way in which they may be used in a subprogram. A dummy data object having INTENT (IN) may not be defined or redefined by the subprogram. A dummy data object having INTENT (OUT) is initially undefined in the subprogram. A dummy data object with INTENT (INOUT) may be referenced or be defined. A dummy data object whose intent is not specified is subject to the limitations of the data entity that is the associated actual argument. That is, a reference to the dummy data object may occur if the actual argument is defined and the dummy data object may be defined if the actual argument is definable.

12.5.2.2 Function Subprogram. A **function subprogram** is a subprogram that has a FUNCTION statement as its first statement.

R1218 *external-subprogram* **is** *procedure-heading*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
procedure-ending

R1219 *procedure-heading* **is** *function-stmt*

or *subroutine-stmt*

R1220 *procedure-ending* **is** *end-function-stmt*
or *end-subroutine-stmt*

R1221 *function-stmt* **is** [*prefix*] FUNCTION *function-name* ■
■ ([*dummy-arg-name-list*]) [RESULT (*result-name*)]

R1222 *prefix* **is** *type-spec* [RECURSIVE]
or RECURSIVE [*type-spec*]

R1223 *end-function-stmt* **is** END [FUNCTION [*function-name*]]

Constraint: For a function subprogram, the *procedure-heading* must be a *function-stmt* and the *procedure-ending* must be an *end-function-stmt.*

Constraint: FUNCTION must be present on the *end-function-stmt* of an internal or module function.

Constraint: An internal function must not contain an ENTRY statement.

Constraint: If a *function-name* is present on the *end-function-stmt*, it must be identical to the *function-name* specified in the *function-stmt.*

The type and type parameters (if any) of the result of the function defined by a function subprogram may be specified by a type specification in the FUNCTION statement or by the function name appearing in a type statement in the declaration part of the function subprogram. It must not be specified both ways. If it is not specified either way, it is determined by the implicit typing rules in force within the function subprogram. If the function result is array-valued or a pointer, this must be specified by specifications of the function name within the function body. The specifications of the function result attribute, the specification of dummy argument attributes, and the information in the procedure heading collectively define the interface of the function (12.3).

The keyword **RECURSIVE** must be present if the function invokes itself, either directly or indirectly.

The name of the function is *function-name.*

If RESULT is specified, the name of the result variable of the function is *result-name* and all occurrences of the function name in *execution-part* statements in the scoping unit are recursive function references. If RESULT is not specified, the result variable is *function-name* and all occurrences of the function name in *execution-part* statements in the scoping unit are references to the result variable. The *result-name* must not appear in any specification statement. 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. The value of the result variable must be defined by the function. If the function result has been declared a pointer, the result variable must be allocated or associated by the function prior to being defined.

An example of a recursive function is:

```
RECURSIVE INTEGER FUNCTION BEST (A, B) RESULT (BST)
   INTEGER A, B
   ...
   BST = BEST (A - 1, B - 1)
END FUNCTION BEST
```

12.5.2.3 Subroutine Subprogram. A **subroutine subprogram** is a subprogram that has a SUBROUTINE statement as its first statement.

R1217 *external-subprogram* **is** *procedure-heading*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]

procedure-ending

R1218 *procedure-heading* **is** *function-stmt*
or *subroutine-stmt*

R1219 *procedure-ending* **is** *end-function-stmt*
or *end-subroutine-stmt*

R1224 *subroutine-stmt* **is** [RECURSIVE] SUBROUTINE *subroutine-name*
[([*dummy-arg-list*])]

R1225 *dummy-arg* **is** *dummy-arg-name*
or *

R1226 *end-subroutine-stmt* **is** END [SUBROUTINE [*subroutine-name*]]

Constraint: For a subroutine subprogram, the *procedure-heading* must be a *subroutine-stmt* and the *procedure-ending* must be an *end-subroutine-stmt*.

Constraint: SUBROUTINE must be present on the END statement of an internal or module subroutine.

Constraint: An internal subroutine must not contain an ENTRY statement.

why not

Constraint: If a *subroutine-name* is present on the *end-subroutine-stmt*, it must be identical to the *subroutine-name* specified in the *subroutine-stmt*.

The keyword RECURSIVE must be present if the subroutine subprogram invokes itself, either directly or indirectly.

12.5.2.4 Instances of a Subprogram. When a function or subroutine defined by a subprogram is invoked, an **Instance** of that subprogram is created.

Each instance has an independent sequence of execution and an independent set of dummy arguments and nonsaved data objects. If an internal procedure or statement function contained in the subprogram is invoked directly from an instance of the subprogram, the created instance of that internal procedure or statement function also has access to the entities of that instance of the host subprogram.

└ how else?

All other entities, including saved data objects, are common to all instances of the subprogram. For example, the value of a saved data object appearing in one instance may have been defined in a previous instance or by a DATA attribute or statement.

? gone, but spelled differently

12.5.2.5 ENTRY Statement.

R1227 *entry-stmt* **is** ENTRY *entry-name* [([*dummy-arg-list*])]

why not

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

Constraint: A *dummy-arg* may be an alternate return indicator only if the ENTRY statement is contained in a subroutine subprogram.

If the ENTRY statement is contained in a function subprogram, an additional function is defined by that subprogram. The name of the function and its result variable is *entry-name*. The characteristics of the function result are specified by specifications of *entry-name*. The dummy arguments of the function are those specified on the ENTRY statement. If the characteristics of the result of the function named on the ENTRY statement are the same as the characteristics of the function named on the FUNCTION statement, their result variables are associated. Otherwise, they are storage associated with the restrictions that they are scalar, that they have type and type parameters permitting storage association, and that they have the same lengths if they are of character type. is * ok?

If the ENTRY statement is contained in a subroutine subprogram, an additional subroutine is defined by that subprogram. The name of the subroutine is *entry-name*. The dummy arguments of the subroutine are those specified on the ENTRY statement. All other dummy arguments are unaffected

12.5.2.6 RETURN Statement.

R1228 *return-stmt* **is** RETURN [*scalar-int-expr*]

Constraint: The *return-stmt* must be contained in the scoping unit of a function or subroutine subprogram.

Constraint: The *scalar-int-expr* is allowed only in the scoping unit of a subroutine subprogram.

Execution of the **RETURN statement** completes execution of the instance of the subprogram in which it appears. If the expression is present and has a value *n* between 1 and the number of asterisks in the dummy argument list, the CALL statement that invoked the subroutine transfers control to the statement identified by the *n*th alternate return specifier in the actual argument list. If the expression is omitted or has a value outside the required range, there is no transfer of control to an alternate return.

Execution of an *end-function-stmt* or *end-subroutine-stmt* is equivalent to executing a RETURN statement with no expression.

12.5.2.7 CONTAINS Statement.

R1229 *contains-stmt* **is** CONTAINS

The **CONTAINS statement** separates the body of a main program, module, or subprogram from any internal or module subprograms it may contain. The CONTAINS statement is not executable.

12.5.2.8 Restrictions on Dummy Arguments Not Present. A dummy argument is **present** in an instance of a subprogram if it is associated with an actual argument and the actual argument either is a dummy argument that is present in the invoking procedure or is not a dummy argument of the invoking procedure. A dummy argument that is not optional must be present. An optional dummy argument that is not present is subject to the following restrictions:

(1) If it is a dummy data object, it must not be referenced or be defined.

(2) If it is a dummy procedure, it must not be invoked.

(3) It must not be supplied as an actual argument corresponding to a nonoptional dummy argument other than the argument of the PRESENT intrinsic function.

(4) It may be supplied as an actual argument corresponding to an optional dummy argument. The optional dummy argument is then also considered not to be associated with an actual argument.

12.5.2.9 Restrictions on Entities Associated with Dummy Arguments. While an entity is associated with a dummy argument, the following restrictions hold:

(1) No action may be taken that affects the value or availability of the entity or any part of it, except through the dummy argument. For example, in

```
SUBROUTINE OUTER
   REAL, POINTER :: A (:)
   ...
   ALLOCATE (A (1:N))
   ...
   CALL INNER (A)
   ...
CONTAINS
   SUBROUTINE INNER (B)
   REAL :: B (:)
   ...
   END SUBROUTINE INNER

   SUBROUTINE SET (C, D)
      REAL, INTENT (OUT) :: C
      REAL, INTENT (IN) :: D
      C = D
```

```
      END SUBROUTINE SET
END SUBROUTINE OUTER
```

an assignment statement such as

```
A (1) = 1.0
```

would not be permitted during the execution of INNER because this would be changing A without using B, but statements such as

```
B (1) = 1.0
```

or

```
CALL SET (B (1), 1.0)
```

would be allowed. Similarly,

```
DEALLOCATE (A)
```

would not be allowed because this affects the availability of A without using B. In this case,

```
DEALLOCATE (B)
```

also would not be permitted, but would be permitted if B were declared ALLOCATABLE.

Note that if there is a partial or complete overlap between the actual arguments associated with two different dummy arguments of the same procedure, the overlapped portions must not be defined or redefined during the execution of the procedure. For example, in

```
CALL SUB (A (1:5), A (3:9))
```

A (3:5) must not be defined or redefined through the first dummy argument because it is part of the argument associated with the second dummy argument and must not be defined or redefined through the second dummy argument because it is part of the argument associated with the first dummy argument. A (1:2) remains definable through the first dummy argument and A (6:9) remains definable through the second dummy argument.

This restriction applies equally to pointer targets. For example, in

```
REAL, DIMENSION(10), TARGET :: A
REAL, DIMENSION(:), POINTER :: B, C
B => A(1:5)
C => A(3:9)
CALL SUB (B, C)
```

B(3:5) cannot be defined because it is part of the argument associated with the second dummy argument. C(1:3) cannot be defined because it is part of the argument associated with the first dummy argument. A(1:2) [which is B(1:2)] remains definable through the first dummy argument and A(6:9) [which is C(4:7)] remains definable through the second dummy argument.

Note that since a dummy argument declared with an intent of IN cannot be used to change the associated actual argument, the associated actual argument remains constant throughout the execution of the procedure.

(2) If any part of the entity is defined through the dummy argument, then at any time during the execution of the procedure, either before or after the definition, it may be referenced only through that dummy argument. For example, in

```
MODULE DATA
   REAL :: W, X, Y, Z
END MODULE DATA
```

```
PROGRAM MAIN
USE DATA
...
CALL INIT (X)
...
END PROGRAM MAIN

SUBROUTINE INIT (V)
   USE DATA
   ...
   READ (KIND (0.0)) V
   ...
END SUBROUTINE INIT
```

variable X must not be directly referenced at any time during the execution of INIT because it is being defined through the dummy argument V. X may be (indirectly) referenced through V. W, Y, and Z may be directly referenced. X may, of course, be directly referenced once execution of INIT is complete.

12.5.3 Definition of Procedures by Means Other Than Fortran. The means other than Fortran by which a procedure may be defined are processor dependent. A reference to such a procedure is made as though it were defined by an external subprogram. The definition of a non-Fortran procedure must not be contained in a Fortran program unit and a Fortran program unit must not be contained in the definition of a non-Fortran procedure. The interface to a non-Fortran procedure may be specified in an interface block.

12.5.4 Statement Function. A statement function is a function defined by a single statement.

R1230 *stmt-function-stmt* **is** *function-name* ([*dummy-arg-name-list*]) = *scalar-expr*

Constraint: The *scalar-expr* may be composed only of constants (literal and named), references to scalar variables and array elements, references to functions and function dummy procedures, and intrinsic operators. If a reference to another statement function appears in *scalar-expr*, its definition must have been provided earlier in the scoping unit.

Constraint: Named constants in *scalar-expr* must have been declared earlier in the scoping unit. If array elements appear in *scalar-expr*, the parent array must have been declared as an array earlier in the scoping unit. If a scalar variable, array element, function reference, or dummy function reference is typed by the implicit typing rules, its appearance in any subsequent type declaration statement must confirm this implied type and the values of any implied type parameters.

Constraint: The *function-name* and each *dummy-arg-name* must be specified, explicitly or implicitly, to be scalar data objects.

Constraint: A given *dummy-arg-name* may appear only once in any *dummy-arg-name-list*.

Constraint: Each scalar variable reference may be either a reference to a dummy argument of the statement function or a reference to a variable within the same scoping unit as the statement function statement.

The dummy arguments have a scope of the statement function statement.

The statement function produces the same result value as an internal function of the form:

```
FUNCTION function-name ( [ dummy-arg-name-list ] )
  function-and-dummy-specifications
  function-name = scalar-expr
END FUNCTION function-name
```

where *function-and-dummy-specifications* are the specifications necessary to cause *function-name* and each *dummy-arg-name* to be given explicitly the same type and type parameters that

those names are given explicitly in the scoping unit containing the statement function. Note, however, that unlike the internal function, the statement function always has an implicit interface. A statement function must not be supplied as a procedure argument.

12.5.5 Overloading Names. Two or more functions may be accessible with the same name in the same scoping unit if any argument list would be appropriate in referencing at most one of them. Similarly, two or more functions may be accessible with the same operator symbol in the same scoping unit, two or more subroutines may be accessible with the same name in the same scoping unit, and two or more subroutines may be accessible as assignments in the same program scope (Section 14). The specific rules on how any two such procedures must differ are given in 14.1.2.3.

13. INTRINSIC PROCEDURES

There are four classes of intrinsic procedures: inquiry function, elemental functions, transformational functions, and subroutines.

13.1 Intrinsic Functions. An **intrinsic function** is an inquiry function, an elemental function, or a transformational function. An **inquiry function** is one whose result depends on the explicit or implicit declarations associated with its principal argument and not on the value of this argument; in fact, the argument value may be undefined. An **elemental function** is one that is specified for scalar arguments, but may be applied to array arguments as described in 13.2. All other intrinsic functions are **transformational functions**; they almost all have one or more array-valued arguments or an array-valued result.

Generic names of intrinsic functions are listed in 13.10. In most cases, generic functions accept arguments of more than one type and the type of the result is the same as the type of the arguments. **Specific names** of intrinsic functions with corresponding generic names are listed in 13.12.

If an intrinsic function is used as an actual argument to a procedure, its specific name must be used and it may be referenced in the procedure only with scalar arguments. If an intrinsic function does not have a specific name, it must not be used as an actual argument.

13.2 Elemental Intrinsic Procedures.

13.2.1 Elemental Intrinsic Function Arguments and Results. If a generic name or a specific name is used to reference an elemental intrinsic function, the shape of the result is the same as the shape of the argument with the greatest rank. If the arguments are all scalar, the result is scalar. For those elemental intrinsic functions that have more than one argument, all arguments must be conformable. In the array-valued case, the values of the elements of the result are the same as would have been obtained if the scalar-valued function had been applied separately to corresponding elements of each argument, in any order.

13.2.2 Elemental Intrinsic Subroutine Arguments. If a generic name is used to reference an elemental intrinsic subroutine, either all actual arguments must be scalar, or all output arguments must be arrays of the same shape and the remaining arguments must be conformable with them. In the case that the output arguments are arrays, the values of the elements of the results are the same as would be obtained if the subroutine with scalar arguments were applied separately to corresponding elements of each argument, in any order.

13.3 Positional Arguments or Argument Keywords. All intrinsic procedures may be invoked with either positional arguments or argument keywords. The descriptions in 13.13 give the keyword names and positional sequence. A keyword is required for an argument only if a preceding optional argument is omitted.

13.4 Argument Presence Inquiry Functions. The inquiry function PRESENT permits an inquiry to be made about the presence of an actual argument associated with a dummy argument.

13.5 Numeric, Mathematical, Character, and Bit Procedures.

13.5.1 Numeric Functions. The elemental functions INT, REAL, DBLE, and CMPLX perform type conversions. The elemental functions AIMAG, CONJG, AINT, ANINT, NINT, ABS, MOD, SIGN, DIM, DPROD, MAX, and MIN perform simple numeric operations.

13.5.2 Mathematical Functions. The elemental functions SQRT, EXP, LOG, LOG10, SIN, COS, TAN, ASIN, ACOS, ATAN, ATAN2, SINH, COSH, and TANH evaluate elementary mathematical functions.

13.5.3 Character Functions. The elemental functions ICHAR, CHAR, LGE, LGT, LLE, LLT, IACHAR, ACHAR, INDEX, VERIFY, ADJUSTL, ADJUSTR, SCAN, and LEN_TRIM perform character operations. The transformational function REPEAT returns repeated concatenations of a character string argument. The transformational function TRIM returns the argument with trailing blanks removed.

13.5.4 Character Inquiry Function. The inquiry function LEN returns the length of a character entity. The value of the argument to this function need not be defined. It is not necessary for a processor to evaluate the argument of this function if the value of the function can be determined otherwise.

13.5.5 Kind Inquiry Functions. The inquiry function KIND returns the kind type parameter value of an integer, real, complex, or character entity. The inquiry function SELECTED_REAL_KIND returns the real kind type parameter value that has at least the decimal precision and exponent range specified by its arguments. The inquiry function SELECTED_INT_KIND returns the integer kind type parameter value that has at least the decimal exponent range specified by its argument.

13.5.6 Logical Functions. The elemental function LOGICAL converts between objects of type logical with different kind type parameter values.

13.5.7 Bit Manipulation and Inquiry Procedures. The bit manipulation procedures consist of a set of ten functions and one subroutine. Logical operations on bits are provided by the functions IOR, IAND, NOT, and IEOR; shift operations are provided by the functions ISHFT and ISHFTC; bit subfields may be referenced by the function IBITS and by the subroutine MVBITS; single-bit processing is provided by the functions BTEST, IBSET, and IBCLR. These procedures are defined by MID-STD 1753 for scalar arguments and are extended in this standard to accept array arguments and to return array-valued results.

For the purposes of these procedures, a bit is defined to be a binary digit w located at position k of a nonnegative integer scalar object based on a model nonnegative integer defined by

$$j = \sum_{k=0}^{s-1} w_k \times 2^k$$

and for which w_k may have the value 0 or 1. An example of a model number compatible with the examples used in 13.7.1 would have $s = 32$, thereby defining a 32-bit integer.

An inquiry function BIT_SIZE is available to determine the parameter s of the model. The value of the argument of this function need not be defined. It is not necessary for a processor to evaluate the argument of this function if the value of the function can be determined otherwise.

Effectively, this model defines an integer object to consist of s bits in an ordered sequence numbered from right to left from 0 to $s-1$. This model is valid only in the context of the use of such an object as the argument or result of one of the bit manipulation procedures. In all other contexts, the model defined for an integer in 13.7.1 applies. In particular, whereas the models are identical for $w_{s-1} = 0$, they do not correspond for $w_{s-1} = 1$ and the interpretation of bits in such objects is processor dependent.

13.6 Transfer Function. The function TRANSFER specifies that the physical representation of the first argument is to be treated as if it were one of the type and type parameters of the second argument with no conversion.

13.7 Numeric Manipulation and Inquiry Functions. The numeric manipulation and inquiry functions are described in terms of a model for the representation and behavior of numbers on a processor. The model has parameters which are determined so as to make the model best fit the machine on which the executable program is executed.

13.7.1 Models for Integer and Real Data. The model set for integer i is defined by:

$$i = s \times \sum_{k=1}^{q} w_k \times r^{k-1}$$

where r is an integer exceeding one, q is a positive integer, each w_k is a nonnegative integer less than r, and s is +1 or −1. The model set for real x is defined by:

$$x = \begin{cases} 0 \ \ or \\ s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k}, \end{cases}$$

where b and p are integers exceeding one; each f_k is a nonnegative integer less than b, except f_1 which is also nonzero; s is +1 or −1; and e is an integer that lies between some integer maximum e_{max} and some integer minimum e_{min} inclusively. For $x = 0$, its exponent e and digits f_k are defined to be zero. The integer parameters r and q determine the set of model integers and the integer parameters b, p, e_{min}, and e_{max} determine the set of model floating point numbers. The parameters of the integer and real models are available for each integer and real data type implemented by the processor. The parameters characterize the set of available numbers in the definition of the model. The floating point manipulation and inquiry functions provide values related to the parameters and other constants related to them. Examples of these functions in this section use the models:

$$i = s \times \sum_{k=1}^{31} w_k \times 2^{k-1}$$

and

$$x = 0 \ \ or \ \ s \times 2^e \times \left[1/2 + \sum_{k=2}^{24} f_k \times 2^{-k} \right], \quad -126 \le e \le 127$$

13.7.2 Numeric Inquiry Functions. The inquiry functions RADIX, DIGITS, MINEXPONENT, MAXEXPONENT, PRECISION, RANGE, HUGE, TINY, and EPSILON return scalar values related to the parameters of the model associated with the type and type parameters of the arguments. The value of the arguments to these functions need not be defined, the shape of pointer arguments may be disassociated, and array arguments need not be allocated.

It is not necessary for a processor to evaluate the arguments of a numeric inquiry function if the value of the function can be determined otherwise.

13.7.3 Floating Point Manipulation Functions. The elemental functions EXPONENT, SCALE, NEAREST, FRACTION, SETEXPONENT, SPACING, and RRSPACING return values related to the components of the model values (13.7.1) associated with the actual values of the arguments.

13.8 Array Intrinsic Functions. The array intrinsic functions perform the following operations on arrays: vector and matrix multiplication, numeric or logical computation that reduces the rank, array structure inquiry, array construction, array manipulation, and geometric location.

13.8.1 The Shape of Array Arguments. The transformational array intrinsic functions operate on each array argument as a whole. The shape of the corresponding actual argument must therefore be defined; that is, the actual argument must be an array section, an assumed-shape array, an explicit-shape array, a pointer that is associated with a target, an allocatable array that has been allocated, or an array-valued expression. It must not be an assumed-size array.

Some of the inquiry intrinsic functions accept array arguments for which the shape need not be defined. Assumed-size arrays may be used as arguments to these functions; they include the function LBOUND and certain references to SIZE and UBOUND.

13.8.2 Mask Arguments. Some array intrinsic functions have an optional MASK argument that is used by the function to select the elements of one or more arguments to be operated on by the function. Any element not selected by the mask need not be defined at the time the function is invoked.

The MASK affects only the value of the function, and does not affect the evaluation, prior to invoking the function, of arguments that are array expressions.

A MASK argument must be of type logical.

13.8.3 Vector and Matrix Multiplication Functions. The matrix multiplication function MATMUL operates on two matrices, or on one matrix and one vector, and returns the corresponding matrix-matrix, matrix-vector, or vector-matrix product. The arguments to MATMUL may be numeric (integer, real, or complex) or logical arrays. On logical matrices and vectors, MATMUL performs Boolean matrix multiplication.

The dot product function DOTPRODUCT operates on two vectors and returns their scalar product. The vectors are of the same type (numeric or logical) as for MATMUL. For logical vectors, DOTPRODUCT returns the Boolean scalar product.

13.8.4 Array Reduction Functions. The array reduction functions SUM, PRODUCT, MAXVAL, MINVAL, COUNT, ANY, and ALL perform numerical, logical, and counting operations on arrays. They may be applied to the whole array to give a scalar result or they may be applied over a given dimension to yield a result of rank reduced by one. By use of a logical mask that is conformable with the given array, the computation may be confined to any subset of the array (e.g., the positive elements).

13.8.5 Array Inquiry Functions. The function ALLOCATED returns a value .TRUE. if the array argument is currently allocated, and returns .FALSE. otherwise. The functions SIZE, SHAPE, LBOUND, and UBOUND return, respectively, the size of the array, the shape, and the lower and upper bounds of the subscripts along each dimension. The size, shape, or bounds must be defined.

The values of the array arguments to these functions need not be defined.

It is not necessary for a processor to evaluate the arguments of an array inquiry function if the value of the function can be determined otherwise.

13.8.6 Array Construction Functions. The functions MERGE, SPREAD, PACK, and UNPACK construct new arrays from the elements of existing arrays. MERGE combines two conformable arrays into one array by an element-wise choice based on a logical mask. SPREAD constructs an array from several copies of an actual argument (SPREAD does this by adding an extra dimension, as in forming a book from copies of one page) RESHAPE produces an array with the same elements and a different shape. PACK and UNPACK respectively gather and scatter the elements of a one-dimensional array from and to positions in another array where the positions are specified by a logical mask.

13.8.7 Array Reshape Function. RESHAPE produces an array with the same elements and a different shape.

13.8.8 Array Manipulation Functions. The functions TRANSPOSE, EOSHIFT, and CSHIFT manipulate arrays. TRANSPOSE performs the matrix transpose operation on a two-dimensional array. The shift functions leave the shape of an array unaltered but shift the positions of the elements parallel to a specified dimension of the array. These shifts are either circular (CSHIFT), in which case elements shifted off one end reappear at the other end, or end-off (EOSHIFT), in which case specified boundary elements are shifted into the vacated positions.

13.8.9 Array Location Functions. The functions MAXLOC and MINLOC return the location (subscripts) of an element of an array that has maximum and minimum values, respectively. By use of an optional logical mask that is conformable with the given array, the reduction may be confined to any subset of the array.

13.8.10 Pointer Association Status Inquiry Functions. The function ASSOCIATED with a single pointer argument returns true if its argument is currently associated with a target, and false if it is currently disassociated. In its two-argument form, it compares the objects. If they refer to the same object, the result is true; otherwise it is false. Two pointers are the same if they are associated with the same target. If the first of the two pointer objects is disassociated, the result if false.

13.9 Intrinsic Subroutines. Intrinsic subroutines are supplied by the processor and have the special definitions given in 13.11 and 13.13. An intrinsic subroutine is referenced by a CALL statement that uses its name explicitly. The name of an intrinsic subroutine must not be used as an actual argument. The effect of a subroutine reference is as specified in 13.13.

13.9.1 Date and Time Subroutines. The subroutines DATE_AND_TIME and SYSTEM_CLOCK return integer data from the date and real-time clock. The time returned is local, but there are facilities for finding out the difference between local time and Coordinated Universal Time.

13.9.2 Pseudorandom Numbers. The subroutine RANDOM returns a pseudorandom number or an array of pseudorandom numbers. The subroutine RANDOMSEED initializes or restarts the pseudorandom number sequence.

13.9.3 Bit Copy Subroutine. The subroutine MVBITS copies a bit field from a specified position in one integer object to a specified position in another.

13.10 Generic Intrinsic Functions. For all of the intrinsic procedures, the arguments shown are the names that must be used for keywords when using the keyword form for actual arguments. For example, a reference to CMPLX may be written in the form CMPLX (TARGET, SOURCE, M) or in the form CMPLX (Y = SOURCE, KIND = M, X = TARGET).

Many of the argument keywords have names that are indicative of their usage. For example:

MOLD	Describes the characteristics of an argument or result
KIND	Describes the KIND of the result
STRING, STRING_A	An arbitrary character string
BACK	Indicates a string scan is to be from right to left (backward)
MASK	A mask that may be applied to the arguments
DIM	A selected dimension of an array argument

13.10.1 Argument Presence Inquiry Function.

PRESENT (A)	Argument presence

13.10.2 Numeric Functions.

ABS (A)	Absolute value
AIMAG (Z)	Imaginary part of a complex number
AINT (A, KIND)	Truncation to whole number
Optional KIND	
ANINT (A, KIND)	Nearest whole number
Optional KIND	
CEILING (A)	Least integer greater than or equal to number
CMPLX (X, Y, KIND)	Conversion to complex type
Optional Y, KIND	
CONJG (Z)	Conjugate of a complex number
DBLE (A)	Conversion to double precision real type
DIM (X, Y)	Positive difference
DPROD (X, Y)	Double precision real product
INT (A, KIND)	Conversion to integer type
Optional KIND	
FLOOR (A)	Greatest integer less than or equal to number
MAX (A1, A2, A3,...)	Maximum value
Optional A3,...	
MIN (A1, A2, A3,...)	Minimum value
Optional A3,...	
MOD (A, P)	Remainder function
MODULO (A, P)	Modulo function
NINT (A, KIND)	Nearest integer
Optional KIND	
REAL (A, KIND)	Conversion to real type
Optional KIND	
SIGN (A, B)	Transfer of sign

13.10.3 Mathematical Functions.

ACOS (X)	Arccosine
ASIN (X)	Arcsine
ATAN (X)	Arctangent
ATAN2 (Y, X)	Arctangent
COS (X)	Cosine
COSH (X)	Hyperbolic cosine
EXP (X)	Exponential
LOG (X)	Natural logarithm
LOG10 (X)	Common logarithm (base 10)
SIN (X)	Sine
SINH (X)	Hyperbolic sine
SQRT (X)	Square root
TAN (X)	Tangent
TANH (X)	Hyperbolic tangent

13.10.4 Character Functions.

ACHAR (I)	Character in given position in ASCII collating sequence
ADJUSTL (STRING)	Adjust left
ADJUSTR (STRING)	Adjust right
CHAR (I, KIND)	Character in given position in processor collating sequence
Optional KIND	

IACHAR (C)	Position of a character in ASCII collating sequence
ICHAR (C)	Position of a character in processor collating sequence
INDEX (STRING, SUBSTRING, BACK) Optional BACK	Starting position of a substring
LEN_TRIM (STRING)	Length without trailing blank characters
LGE (STRING_A, STRING_B)	Lexically greater than or equal
LGT (STRING_A, STRING_B)	Lexically greater than
LLE (STRING_A, STRING_B)	Lexically less than or equal
LLT (STRING_A, STRING_B)	Lexically less than
REPEAT (STRING, NCOPIES)	Repeated concatenation
SCAN (STRING, SET, BACK) Optional BACK	Scan a string for a character in a set
TRIM (STRING)	Remove trailing blank characters
VERIFY (STRING, SET, BACK) Optional BACK	Verify the set of characters in a string

13.10.5 Character Inquiry Functions.

LEN (STRING)	Length of a character entity

13.10.6 Kind Inquiry Function.

KIND (X)	Kind parameter value
SELECTED_INT_KIND (R)	Integer kind type parameter value, given range
SELECTED_REAL_KIND (P, R)	Real kind type parameter value, given precision and range

13.10.7 Logical Functions.

LOGICAL (L, KIND) Optional KIND	Convert between objects of type logical with different kind type parameters

13.10.8 Numeric Inquiry Functions.

DIGITS (X)	Number of significant digits in the model
EPSILON (X)	Number that is almost negligible compared to one
HUGE (X)	Largest number in the model
MAXEXPONENT (X)	Maximum exponent in the model
MINEXPONENT (X)	Minimum exponent in the model
PRECISION (X)	Decimal precision
RADIX (X)	Base of the model
RANGE (X)	Decimal exponent range
TINY (X)	Smallest number in the model

13.10.9 Bit Inquiry Functions.

BIT_SIZE (I)	Number of bits in the model

13.10.10 Bit Manipulation Functions.

BTEST (I, POS)	Bit testing
IAND (I, J)	Logical AND
IBCLR (I, POS)	Clear bit
IBITS (I, POS, LEN)	Bit extraction
IBSET (I, POS)	Set bit

Univac FLD or BEXTS accessor would be more economical

why not just extend .AND. etc?

IEOR (I, J)	Exclusive OR
IOR (I, J)	Inclusive OR
ISHFT (I, SHIFT)	Logical shift
ISHFTC (I, SHIFT, SIZE) Optional SIZE	Circular shift
NOT (I)	Logical complement

13.10.11 Transfer Function.

TRANSFER (SOURCE, MOLD, SIZE) Optional SIZE	Treat first argument as if of type of second argument

13.10.12 Floating-point Manipulation Functions.

EXPONENT (X)	Exponent part of a model number
FRACTION (X)	Fractional part of a number
NEAREST (X, S)	Nearest different processor number in given direction
RRSPACING (X)	Reciprocal of the relative spacing of model numbers near given number
SCALE (X, I)	Multiply a real by its base to an integer power
SETEXPONENT (X, I)	Set exponent part of a number
SPACING (X)	Absolute spacing of model numbers near given number

13.10.13 Vector and Matrix Multiply Functions.

DOTPRODUCT (VECTOR_A, VECTOR_B)	Dot product of two rank-one arrays
MATMUL (MATRIX_A, MATRIX_B)	Matrix multiplication

13.10.14 Array Reduction Functions.

ALL (MASK, DIM) Optional DIM	True if all values are true
ANY (MASK, DIM) Optional DIM	True if any value is true
COUNT (MASK, DIM) Optional DIM	Number of true elements in an array
MAXVAL (ARRAY, DIM, MASK) Optional DIM, MASK	Maximum value in an array
MINVAL (ARRAY, DIM, MASK) Optional DIM, MASK	Minimum value in an array
PRODUCT (ARRAY, DIM, MASK) Optional DIM, MASK	Product of array elements
SUM (ARRAY, DIM, MASK) Optional DIM, MASK	Sum of array elements

13.10.15 Array Inquiry Functions.

ALLOCATED (ARRAY)	Array allocation status
LBOUND (ARRAY, DIM) Optional DIM	Lower dimension bounds of an array
SHAPE (SOURCE)	Shape of an array or scalar
SIZE (ARRAY, DIM) Optional DIM	Total number of elements in an array
UBOUND (ARRAY, DIM) Optional DIM	Upper dimension bounds of an array

13.10.16 Array Construction Functions.

MERGE (TSOURCE, FSOURCE, MASK)	Merge under mask
PACK (ARRAY, MASK, VECTOR)	Pack an array into an array of rank one under a mask
Optional VECTOR	
SPREAD (SOURCE, DIM, NCOPIES)	Replicates array by adding a dimension
UNPACK (VECTOR, MASK, FIELD)	Unpack an array of rank one into an array under a mask

13.10.17 Array Reshape Function.

RESHAPE (SHAPE, SOURCE, PAD, ORDER)	Reshape an array
Optional PAD, ORDER	

13.10.18 Array Manipulation Functions.

CSHIFT (ARRAY, DIM, SHIFT)	Circular shift
EOSHIFT (ARRAY, DIM, SHIFT, BOUNDARY)	End-off shift
Optional BOUNDARY	
TRANSPOSE (MATRIX)	Transpose of an array of rank two

13.10.19 Array Location Functions.

MAXLOC (ARRAY, MASK)	Location of a maximum value in an array
Optional MASK	
MINLOC (ARRAY, MASK)	Location of a minimum value in an array
Optional MASK	

13.10.20 Pointer Association Status Inquiry Functions.

ASSOCIATED (POINTER, TARGET)	Association status or comparison
Optional TARGET	

13.11 Intrinsic Subroutines.

DATE_AND_TIME (ALL, COUNT, MSECOND, SECOND, MINUTE, HOUR, DAY, MONTH, YEAR, ZONE)	Obtain date and time
Optional ALL, COUNT, MSECOND, SECOND, MINUTE, HOUR, DAY, MONTH, YEAR, ZONE	
MVBITS (FROM, FROMPOS, LEN, TO, TOPOS)	Copies bits from one integer to another
RANDOM (HARVEST)	Returns pseudorandom number
RANDOMSEED (SIZE, PUT, GET)	Initializes or restarts the pseudorandom number generator
Optional SIZE, PUT, GET	
SYSTEM_CLOCK (COUNT, COUNT_RATE, COUNT_MAX)	Obtain data from the system clock
Optional COUNT, COUNT_RATE, COUNT_MAX	

13.12 Specific Names for Intrinsic Functions.

	Specific Name	*Generic Name*	*Argument Type*
	ABS (A)	ABS (A)	default real
	ACOS (X)	ACOS (X)	default real
	AIMAG (Z)	AIMAG (Z)	default complex
	AINT (A)	AINT (A)	default real
	ALOG (X)	LOG (X)	default real
	ALOG10 (X)	LOG10 (X)	default real
•	AMAX0 (A1,A2,A3,...) Optional A3,...	REAL (MAX (A1, A2,A3,...)) Optional A3,...	default integer
•	AMAX1 (A1,A2,A3,...) Optional A3,...	MAX (A1, A2,A3,...) Optional A3,...	default real
•	AMIN0 (A1,A2,A3,...) Optional A3,...	REAL (MIN (A1, A2,A3,...)) Optional A3,...	default integer
•	AMIN1 (A1,A2,A3,...) Optional A3,...	MIN (A1, A2,A3,...) Optional A3,...	default real
	AMOD (A,P)	MOD (A,P)	default real
	ANINT (A)	ANINT (A)	default real
	ASIN (X)	ASIN (X)	default real
	ATAN (X)	ATAN (X)	default real
	ATAN2 (Y,X)	ATAN2 (Y,X)	default real
	CABS (A)	ABS (A)	default complex
	CCOS (X)	COS (X)	default complex
	CEXP (X)	EXP (X)	default complex
•	CHAR (I)	CHAR (I)	default integer
	CLOG (X)	LOG (X)	default complex
	CONJG (Z)	CONJG (Z)	default complex
	COS (X)	COS (X)	default real
	COSH (X)	COSH (X)	default real
	CSIN (X)	SIN (X)	default complex
	CSQRT (X)	SQRT (X)	default complex
	DABS (A)	ABS (A)	double precision real
	DACOS (X)	ACOS (X)	double precision real
	DASIN (X)	ASIN (X)	double precision real
	DATAN (X)	ATAN (X)	double precision real
	DATAN2 (Y,X)	ATAN2 (Y,X)	double precision real
	DCOS (X)	COS (X)	double precision real
	DCOSH (X)	COSH (X)	double precision real
	DDIM (X,Y)	DIM (X,Y)	double precision real
	DEXP (X)	EXP (X)	double precision real
	DIM (X,Y)	DIM (X,Y)	default real
	DINT (A)	AINT (A)	double precision real
	DLOG (X)	LOG (X)	double precision real
	DLOG10 (X)	LOG10 (X)	double precision real
•	DMAX1 (A1,A2,A3,...) Optional A3,...	MAX (A1,A2,A3,...) Optional A3,...	double precision real
•	DMIN1 (A1,A2,A3,...) Optional A3,...	MIN (A1,A2,A3,...) Optional A3,...	double precision real
	DMOD (A,P)	MOD (A,P)	double precision real
	DNINT (A)	ANINT (A)	double precision real
	DPROD (X,Y)	DPROD (X,Y)	default real
	DSIGN (A,B)	SIGN (A,B)	double precision real

DSIN (X)	SIN (X)	double precision real
DSINH (X)	SINH (X)	double precision real
DSQRT (X)	SQRT (X)	double precision real
DTAN (X)	TAN (X)	double precision real
DTANH (X)	TANH (X)	double precision real
EXP (X)	EXP (X)	default real
· FLOAT (A)	REAL (A)	default integer
IABS (A)	ABS (A)	default integer
· ICHAR (C)	ICHAR (C)	character
IDIM (X,Y)	DIM (X,Y)	default integer
· IDINT (A)	INT (A)	double precision real
IDNINT (A)	NINT (A)	double precision real
· IFIX (A)	INT (A)	default real
INDEX (S,T)	INDEX (S,T)	character
· INT (A)	INT (A)	default real
ISIGN (A,B)	SIGN (A,B)	default integer
LEN (S)	LEN (S)	character
· LGE (S,T)	LGE (S,T)	character
· LGT (S,T)	LGT (S,T)	character
· LLE (S,T)	LLE (S,T)	character
· LLT (S,T)	LLT (S,T)	character
· MAX0 (A1,A2,A3,...) Optional A3,...	MAX (A1,A2,A3,...) Optional A3,...	default integer
· MAX1 (A1,A2,A3,...) Optional A3,...	INT (MAX (A1,A2,A3,...)) Optional A3,...	default real
· MIN0 (A1,A2,A3,...) Optional A3,...	MIN (A1,A2,A3,...) Optional A3,...	default integer
· MIN1 (A1,A2,A3,...) Optional A3,...	INT (MIN (A1,A2,A3,...)) Optional A3,...	default real
MOD (A,P)	MOD (A,P)	default integer
NINT (A)	NINT (A)	default real
· REAL (A)	REAL (A)	default integer
SIGN (A,B)	SIGN (A,B)	default real
SIN (X)	SIN (X)	default real
SINH (X)	SINH (X)	default real
· SNGL (A)	REAL (A)	double precision real
SQRT (X)	SQRT (X)	default real
TAN (X)	TAN (X)	default real
TANH (X)	TANH (X)	default real

· These specific intrinsic function names must not be used as an actual argument.

13.13 Specifications of the Intrinsic Procedures. This section contains detailed specifications of all the intrinsic procedures in alphabetical order.

13.13.1 ABS (A).

Description. Absolute value.

Class. Elemental function.

Argument. A must be of type integer, real, or complex.

Result Type and Type Parameter. The same as A except that if A is complex, the result is real.

Result Value. If A is of type integer or real, the value of the result is |A|; if A is complex with value (x, y), the result is equal to a processor-dependent approximation to $\sqrt{x^2+y^2}$.

Example. ABS ((3.0, 4.0)) has the value 5.0 (approximately).

13.13.2 ACHAR (I).

Description. Returns the character in a specified position of the ASCII collating sequence. It is the inverse of the IACHAR function.

Class. Elemental function.

Argument. I must be of type integer.

Result Type and Type Parameter. Character of length one with kind type parameter value KIND ('A').

Result Value. If I has a value in the range $0 \leq I \leq 127$, the result is the character in position I of the ASCII collating sequence; otherwise, the result is processor dependent. If the processor is not capable of representing both upper and lower case letters and I corresponds to an ASCII letter in a case that the processor is not capable of representing, the result is the letter in the case that the processor is capable of representing. ACHAR (IACHAR (C)) must have the value C for any character C capable of representation in the processor.

Example. ACHAR (88) has the value 'X'.

13.13.3 ACOS (X).

Description. Arccosine (inverse cosine) function.

Class. Elemental function.

Argument. X must be of type real with a value that satisfies the inequality $|X| \leq 1$.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to arccos(X), expressed in radians. It lies in the range $0 \leq \text{ACOS}(X) \leq \pi$.

Example. ACOS (0.54030231) has the value 1.0 (approximately).

13.13.4 ADJUSTL (STRING).

Description. Adjust to the left, removing leading blanks and inserting trailing blanks.

Class. Elemental function.

Argument. STRING must be of type character.

Result Type. Character of the same length and kind type parameter as STRING.

Result Value. The value of the result is the same as STRING except that any leading blanks have been deleted and the same number of trailing blanks have been inserted.

Example. ADJUSTL (' WORD') has the value 'WORD '.

13.13.5 ADJUSTR (STRING).

Description. Adjust to the right, removing trailing blanks and inserting leading blanks.

Class. Elemental function.

Argument. STRING must be of type character.

Result Type. Character of the same length and kind type parameter as STRING.

Result Value. The value of the result is the same as STRING except that any trailing blanks have been deleted and the same number of leading blanks have been inserted.

Example. ADJUSTR ('WORD ') has the value ' WORD'.

13.13.6 AIMAG (Z).

Description. Imaginary part of a complex number.

Class. Elemental function.

Argument. Z must be of type complex.

Result Type and Type Parameter. Real with the same type parameter as Z.

Result Value. If Z has the value (x, y), the result has value y.

Example. AIMAG ((2.0, 3.0)) has the value 3.0.

13.13.7 AINT (A, KIND).

Optional Argument. KIND

Description. Truncation to a whole number.

Class. Elemental function.

Arguments.

A	must be of type real.
KIND (optional)	must be a scalar integer constant expression.

Result Type and Type Parameter. The result is of type real. If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of A.

Result Value. If $|A| < 1$, AINT (A) has the value 0; if $|A| \geq 1$, AINT (A) has a value equal to the largest integer that does not exceed the magnitude of A and whose sign is the same as the sign of A.

Example. AINT (2.783) has the value 2.0.

13.13.8 ALL (MASK, DIM).

Optional Argument. DIM

Description. Determine whether all values are true in MASK along dimension DIM.

Class. Transformational function.

Arguments.

MASK	must be of type logical. It must not be scalar.
DIM (optional)	must be scalar and of type integer with value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of MASK.

Result Type and Shape. The result is of type default logical. It is scalar if DIM is absent or MASK has rank one; otherwise, the result is an array of rank $n-1$ and of shape $(d_1, d_2, ..., d_{\text{DIM}-1}, d_{\text{DIM}+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of MASK.

Result Value.

Case (i): The result of ALL (MASK) has the value .TRUE. if all elements of MASK are true or if MASK has size zero, and the result has value .FALSE. if any element of MASK is false.

Case (ii): If MASK has rank one, ALL (MASK, DIM) has a value equal to that of ALL (MASK). Otherwise, the value of element $(s_1, s_2, ..., s_{\text{DIM}-1}, s_{\text{DIM}+1}, ..., s_n)$ of ALL (MASK, DIM) is equal to ALL (MASK $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$).

Examples.

Case (i): The value of ALL ((/ .TRUE., .FALSE., .TRUE. /)) is .FALSE.

Case (ii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ and C is the array $\begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$, then ALL (B .NE. C, DIM = 1) is (/ .TRUE., .FALSE., .FALSE. /) and ALL (B .NE. C, DIM = 2) is (/ .FALSE., .FALSE. /).

13.13.9 ALLOCATED (ARRAY).

Description. Indicate whether or not an allocatable array is currently allocated.

Class. Inquiry function.

Argument. ARRAY must be an allocatable array.

Result Type, Type Parameter, and Shape. Default logical scalar.

Result Value. The result has the value .TRUE. if ARRAY is currently allocated and has the value .FALSE. otherwise.

13.13.10 ANINT (A, KIND).

Optional Argument. KIND

Description. Nearest whole number.

Class. Elemental function.

Arguments.

A	must be of type real.
KIND (optional)	must be a scalar integer constant expression.

Result Type and Type Parameter. The result is of type real. If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of A.

Result Value. If $A > 0$, ANINT (A) has the value AINT (A + 0.5); if $A \le 0$, ANINT (A) has the value AINT (A – 0.5).

Example. ANINT (2.783) has the value 3.0

13.13.11 ANY (MASK, DIM).

Optional Argument. DIM

Description. Determine whether any value is true in MASK along dimension DIM.

Class. Transformational function.

Arguments.

MASK	must be of type logical. It must not be scalar.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \le \text{DIM} \le n$, where n is the rank of MASK.

Result Type and Shape. The result is of type default logical. It is scalar if DIM is absent or MASK has rank one; otherwise, the result is an array of rank $n-1$ and of shape $(d_1, d_2, ..., d_{\text{DIM}-1}, d_{\text{DIM}+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of MASK.

Result Value.

Case (i): The result of ANY (MASK) has the value .TRUE. if any element of MASK is true and has the value .FALSE. if no elements are true or if MASK has size zero.

Case (ii): If MASK has rank one, ANY (MASK, DIM) has a value equal to that of ANY (MASK). Otherwise, the value of element $(s_1, s_2, ..., s_{\text{DIM}-1}, s_{\text{DIM}+1}, ..., s_n)$ of ANY (MASK, DIM) is equal to ANY (MASK $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$.

Examples.

Case (i): The value of ANY ((/ .TRUE., .FALSE., .TRUE. /)) is .TRUE.

Case (ii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ and C is the array $\begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$, ANY (B .NE. C, DIM = 1) is (/ .TRUE., .FALSE., .TRUE. /) and ANY (B .NE. C, DIM = 2) is (/ .TRUE., .TRUE. /).

13.13.12 ASIN (X).

Description. Arcsine (inverse sine) function.

Class. Elemental function.

Argument. X must be of type real. Its value must satisfy the inequality $|X| \leq 1$.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to arcsin(X), expressed in radians. It lies in the range $-\pi/2 \leq \text{ASIN}(X) \leq \pi/2$.

Example. ASIN (0.84147098) has the value 1.0 (approximately).

13.13.13 ASSOCIATED (POINTER, TARGET).

Optional Argument. TARGET

Description. Returns the association status of its pointer argument or indicates the pointer is associated with the target.

Class. Inquiry function.

Arguments.

POINTER must be a pointer and may be of any type.

TARGET (optional) must be a pointer or target.

Result Type. The result is of type default logical.

Case (i): If TARGET is absent, the result is true if POINTER is currently associated with a target and false if it is not.

Case (ii): If TARGET is present, the result is true if the POINTER is currently associated with TARGET and false if it is not. If POINTER is disassociated, the result is false.

Example. ASSOCIATED (CURRENT, HEAD) is true if CURRENT points to the target HEAD.

13.13.14 ATAN (X).

Description. Arctangent (inverse tangent) function.

Class. Elemental function.

Argument. X must be of type real.

Y (optional)

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to arctan(X), expressed in radians, that lies in the range $-\pi/2 \leq \text{ATAN}(X) \leq \pi/2$.

Example. ATAN (1.5574077) has the value 1.0 (approximately).

13.13.15 ATAN2 (Y, X).

Description. Arctangent (inverse tangent) function. The result is the principal value of the argument of the nonzero complex number (X, Y).

Class. Elemental function.

Arguments.

Y	must be of type real.
X	must be of the same type and kind type parameter as Y. If Y has the value zero, X must not have the value zero.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to the argument of the complex number (X, Y), expressed in radians. It lies in the range $-\pi < \text{ATAN2}(Y, X) \leq \pi$ and is equal to a processor-dependent approximation to a value of arctan(Y/X) if $X \neq 0$. If $Y > 0$, the result is positive. If $Y = 0$, the result is zero if $X > 0$ and the result is π if $X < 0$. If $Y < 0$, the result is negative. If $X = 0$, the absolute value of the result is $\pi/2$.

Example. ATAN2 (1.5574077, 1.0) has the value 1.0 (approximately).

13.13.16 BIT_SIZE (I).

Description. Returns the number of bits *s* defined by the model.

Class. Inquiry function.

Argument. I must be of type integer.

Result Type and Type Parameter. Same as I.

Result Value. The result has the value of the number of bits *s* in the model integer defined for bit manipulation contexts in 13.5.7.

Example. BIT_SIZE (1) has the value 32 if *s* in the model is 32.

13.13.17 BTEST (I, POS).

Description. Tests a bit of an integer value.

Class. Elemental function.

Arguments.

I	must be of type integer.
POS	must be of type integer. It must be nonnegative and be less than BIT_SIZE (I).

Result Type. The result is of type default logical.

Result Value. The result has the value .TRUE. if bit POS of I has the value 1 and has the value .FALSE. if bit POS of I has the value 0.

Example. BTEST (8, 3) has the value .TRUE.

13.13.18 CEILING (A).

Description. Returns the least integer greater than or equal to its argument.

Class. Elemental function.

Argument. A must of type real.

Result Type and Type Parameter. Default integer.

Result Value. The result has a value equal to the least integer greater than or equal to A. The result is undefined if the processor cannot rerpresent this value in the default integer type.

Example. CEILING (3.7) has the value 4. CEILING (–3.7) has the value –3.

13.13.19 CHAR (I, KIND).

Optional Argument. KIND

Description. Returns the character in a given position of the processor collating sequence. It is the inverse of the function ICHAR.

Class. Elemental function.

Arguments.

I	must be of type integer with a value in the range $0 \le I \le n-1$, where n is the number of characters in the collating sequence.
KIND (optional)	must be a scalar integer constant expression.

Result Type and Type Parameters. Character of length one. If KIND is present, the kind type parameter is that specilfied by KIND; otherwise, the kind type parameter is that of default character type.

Result Value. The result is the character in position I of the processor collating sequence. ICHAR (CHAR (I)) must have the value I for $0 \le I \le n-1$ and CHAR (ICHAR (C)) must have the value C for any character C capable of representation in the processor.

Example. CHAR (88) has the value 'X' on a processor using the ASCII collating sequence.

13.13.20 CMPLX (X, Y, KIND).

Optional Arguments. Y, KIND

Description. Convert to complex type.

Class. Elemental function.

Arguments.

X	must be of type integer, real, or complex.
Y (optional)	must be of type integer or real. It must not be present if X is of type complex.
KIND (optional)	must be a scalar integer constant expression.

Result Type and Type Parameter. The result is of type complex. If KIND is present, the type parameter is that specified by KIND; otherwise, the type parameter is that of default real type.

Result Value. If Y is absent and X is not complex, it is as if Y were present with the value zero. If X is complex, it is as if Y were present with the value AIMAG (X). CMPLX (X, Y, KIND) has the complex value whose real part is REAL (X, KIND) and whose imaginary part is REAL (Y, KIND).

Example. CMPLX (–3) has the value (–3.0, 0.0).

13.13.21 CONJG (Z).

Description. Conjugate of a complex number.

Class. Elemental function.

Argument. Z must be of type complex.

Result Type and Type Parameter. Same as Z.

Result Value. If Z has the value (x, y), the result has the value $(x, -y)$.

Example. CONJG ((2.0, 3.0)) has the value (2.0, –3.0).

13.13.22 COS (X).

Description. Cosine function.

Class. Elemental function.

Argument. X must be of type real or complex.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to cos(X). If X is of type real, it is regarded as a value in radians. If X is of type complex, its real part is regarded as a value in radians.

Example. COS (1.0) has the value 0.54030231 (approximately).

13.13.23 COSH (X).

Description. Hyperbolic cosine function.

Class. Elemental function.

Argument. X must be of type real.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to cosh(X).

Example. COSH (1.0) has the value 1.5430806 (approximately).

13.13.24 COUNT (MASK, DIM).

Optional Argument. DIM

Description. Count the number of true elements of MASK along dimension DIM.

Class. Transformational function.

Arguments.

MASK	must be of type logical. It must not be scalar.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \le \text{DIM} \le n$, where n is the rank of MASK.

Result Type, Type Parameter, and Shape. The result is of type default integer. It is scalar if DIM is absent or MASK has rank one; otherwise, the result is an array of rank $n-1$ and of shape $(d_1, d_2, ..., d_{\text{DIM}-1}, d_{\text{DIM}+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of MASK.

Result Value.

Case (i): The result of COUNT (MASK) has a value equal to the number of true elements of MASK or has the value zero if MASK has size zero.

Case (ii): If MASK has rank one, COUNT (MASK, DIM) has a value equal to that of COUNT (MASK). Otherwise, the value of element $(s_1, s_2, ..., s_{\text{DIM}-1}, s_{\text{DIM}+1}, ..., s_n)$ of COUNT (MASK, DIM) is equal to COUNT (MASK $(s_1, s_2, ..., s_{\text{DIM}-1}, s_{\text{DIM}+1}, ..., s_n)$).

Examples.

Case (i): The value of COUNT ((/ .TRUE., .FALSE., .TRUE. /)) is 2.

Case (ii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ and C is the array $\begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$, COUNT (B .NE. C, DIM = 1) is (/ 2, 0, 1 /) and COUNT (B .NE. C, DIM = 2) is (/ 1, 2 /).

13.13.25 CSHIFT (ARRAY, DIM, SHIFT).

Description. Perform a circular shift on an array expression of rank one or perform circular shifts on all the complete rank one sections along a given dimension of an array expression of rank two or greater. Elements shifted out at one end of a section are shifted in at the other end. Different sections may be shifted by different amounts and in different directions.

Class. Transformational function.

Arguments.

ARRAY	may be of any type. It must not be scalar.
DIM	must be a scalar and of type integer with a value in the range $1 \le \text{DIM} \le n$, where n is the rank of ARRAY.
SHIFT	must be of type integer and must be scalar if ARRAY has rank one; otherwise, it must be scalar or of rank $n-1$ and of shape $(d_1, d_2, ..., d_{\text{DIM}-1}, d_{\text{DIM}+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of ARRAY.

Result Type, Type Parameter, and Shape. The result is of the type and type parameters of ARRAY, and has the shape of ARRAY.

Result Value.

Case (i): If ARRAY has rank one, the result is obtained by applying the |SHIFT| circular shifts to ARRAY in the direction indicated by the sign of SHIFT. If SHIFT has the value 1, element i of the result is ARRAY (i+1) for $i = 1, 2, ..., m-1$ and element m of the result is ARRAY (1) where m is the size of ARRAY. If SHIFT is positive, the result is equivalent to SHIFT applications of CSHIFT with SHIFT=1. If SHIFT has the value –1, element i of the result is ARRAY (i–1) for $i = 2, 3, ..., m$ and element 1 of the result is ARRAY (m). If SHIFT is negative, the result is equivalent to –SHIFT applications of CSHIFT with SHIFT=–1.

Case (ii): If ARRAY has rank greater than one, section $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$ of the result has a value equal to CSHIFT (ARRAY $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$, 1, sh), where sh is SHIFT or SHIFT $(s_1, s_2, ..., s_{\text{DIM}-1}, s_{\text{DIM}+1}, ..., s_n)$.

Examples.

Case (i): If V is the array (/ 1, 2, 3, 4, 5, 6 /), the effect of shifting V circularly to the left by two positions is achieved by CSHIFT (V, DIM=1, SHIFT=2) which has the value (/ 3, 4, 5, 6, 1, 2 /); CSHIFT (V, DIM=1, SHIFT=–2) achieves a circular shift to the right by two positions and has the value (/ 5, 6, 1, 2, 3, 4 /).

Case (ii): The rows of an array of rank two may all be shifted by the same amount or by different amounts. If M is the array $\begin{bmatrix} A & B & C \\ A & B & C \\ A & B & C \end{bmatrix}$, the value of CSHIFT (M, DIM=2, SHIFT=–1) is $\begin{bmatrix} C & A & B \\ C & A & B \\ C & A & B \end{bmatrix}$, and the value of CSHIFT (M, DIM=2, SHIFT=(/ –1, 1, 0 /)) is $\begin{bmatrix} C & A & B \\ B & C & A \\ A & B & C \end{bmatrix}$.

13.13.26 DATE_AND_TIME (ALL, COUNT, MSECOND, SECOND, MINUTE, HOUR, DAY, MONTH, YEAR, ZONE).

Optional Arguments. ALL, COUNT, MSECOND, SECOND, MINUTE, HOUR, DAY, MONTH, YEAR, ZONE

Description. Returns integer data from the date and real-time clock available to the processor. All values returned must refer to the same instant in time.

Class. Subroutine.

Arguments.

ALL (optional) must be of type default integer and rank one. Its size must be at least 9. The values returned in ALL are as for the remaining 9 arguments, taken in order.

COUNT (optional) must be scalar and of type default integer. It is set to a processor-dependent value based on the current value of the basic clock or to –HUGE (0) if there is no clock. The processor-dependent value is incremented by one for each clock count until the value COUNT_MAX (as returned by subroutine SYSTEM_CLOCK) is reached and is reset to zero at the next count. It lies in the range 0 to COUNT_MAX if there is a clock.

MSECOND (optional) must be scalar and of type default integer. It is set to the millisecond part of the local time, or to –HUGE (0) if there is no clock. It lies in the range 0 to 999 if there is a clock.

SECOND (optional) must be scalar and of type default integer. It is set to the second part of the local time, or to –HUGE (0) if there is no clock. It lies in the range 0 to 59 if there is a clock.

MINUTE (optional) must be scalar and of type default integer. It is set to the minute part of the local time, or to –HUGE (0) if there is no clock. It lies in the range 0 to 59 if there is a clock.

HOUR (optional) must be scalar and of type default integer. It is set to the hour part of the local time, or to –HUGE (0) if there is no clock. It lies in the range 0 to 23 if there is a clock.

DAY (optional) must be scalar and of type default integer. It is set to the day of the month, or to –HUGE (0) if there is no date available. It lies in the range 1 to 31 if there is a date available.

MONTH (optional) must be scalar and of type default integer. It is set to the month of the year, or to –HUGE (0) if there is no date available. It lies in the range 1 to 12 if there is a date available.

YEAR (optional) must be scalar and of type default integer. It is set to the year according to the Gregorian calendar (e.g. 1988), or to –HUGE (0) if there is no date available.

ZONE (optional) must be scalar and of type default integer. It is set to the number of minutes that local time is in advance of Coordinated Universal Time, or to –HUGE (0) if this information is not available.

Example.

```
CALL DATE_AND_TIME (ZONE = HERE)
```

will assign the value –300 to the variable HERE if the local time is 5 hours behind Coordinated Universal Time.

13.13.27 DBLE (A).

Description. Convert to double precision real type.

Class. Elemental function.

Argument. A must be of type integer, real, or complex.

Result Type and Type Parameter. Double precision real.

Result Value.

Case (i): If A is of type double precision real, DBLE (A) = A.

Case (ii): If A is of type integer or real, the result is as much precision of the significant part of A as a double precision real datum can contain.

Case (iii): If A is of type complex, the result is as much precision of the significant part of the real part of A as a double precision real datum can contain.

Example. DBLE (–3) has the value –3.0D0.

13.13.28 DIGITS (X).

Description. Returns the number of significant digits in the model representing numbers of the same type and type parameter as the argument.

Class. Inquiry function.

Argument. X must be of type integer or real. It may be scalar or array valued.

Result Type, Type Parameter, and Shape. Default integer scalar.

Result Value. The result has the value q if X is of type integer and p if X is of type real, where q and p are as defined in 13.7.1 for the model representing numbers of the same type and type parameter as X.

Example. DIGITS (X) has the value 24 for real X whose model is as at the end of 13.7.1.

13.13.29 DIM (X, Y).

Description. The difference X–Y if it is positive; otherwise zero.

Class. Elemental function.

Arguments.

X	must be of type integer or real.
Y	must be of the same type and type parameter as X.

Result Type and Type Parameter. Same as X.

Result Value. The value of the result is X–Y if X > Y and zero otherwise.

Example. DIM (–3.0, 2.0) has the value 0.0.

13.13.30 DOTPRODUCT (VECTOR_A, VECTOR_B).

Description. Performs dot-product multiplication of numeric or logical vectors.

Class. Transformational function.

Arguments.

VECTOR_A	must be of numeric type (integer, real, or complex) or of logical type. It must be array valued and of rank one.
VECTOR_B	must be of numeric type if VECTOR_A is of numeric type or of type logical if VECTOR_A is of type logical. It must be array valued and of rank one. It must be of the same size as VECTOR_A.

Result Type and Type Parameter. If the arguments are of numeric type, the type and type parameter of the result are those of the expression VECTOR_A * VECTOR_B determined by the types of the arguments according to 7.1.4. If the arguments are of type logical, the result is of type logical. The result is scalar.

Result Value.

Case (i): If VECTOR_A is of type integer or real, the result has the value SUM (VECTOR_A*VECTOR_B). If the vectors have size zero, the result has the value zero.

Case (ii): If VECTOR_A is of type complex, the result has the value SUM (CONJG (VECTOR_A)*VECTOR_B). If the vectors have size zero, the result has the value zero.

Case (iii): If VECTOR_A is of type logical, the result has the value ANY (VECTOR_A .AND. VECTOR_B). If the vectors have size zero, the result has the value .FALSE.

Example. DOTPRODUCT ((/ 1, 2, 3 /), (/ 2, 3, 4 /)) has the value 20.

13.13.31 DPROD (X, Y).

Description. Double precision real product.

Class. Elemental function.

Arguments.

X	must be of type default real.
Y	must be of type default real.

Result Type and Type Parameters. Double precision real.

Result Value. The value of the result is DBLE (X) * DBLE (Y).

Example. DPROD (–3.0, 2.0) has the value –6.0D0.

13.13.32 EOSHIFT (ARRAY, DIM, SHIFT, BOUNDARY).

Optional Argument. BOUNDARY

Description. Perform an end-off shift on an array expression of rank one or perform end-off shifts on all the complete rank-one sections along a given dimension of an array expression of rank two or greater. Elements are shifted off at one end of a section and copies of a boundary value are shifted in at the other end. Different sections may have different boundary values and may be shifted by different amounts and in different directions.

Class. Transformational function.

Arguments.

ARRAY	may be of any type. It must not be scalar.
DIM	must be scalar and of type integer with a value in the range $1 \le \text{DIM} \le n$, where n is the rank of ARRAY.
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 $(d_1, d_2, ..., d_{\text{DIM}-1}, d_{\text{DIM}+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of ARRAY.
BOUNDARY (optional)	must be of the same type and type parameters as ARRAY and must be scalar if ARRAY has rank one; otherwise, it must be either scalar or of rank $n-1$ and of shape $(d_1, d_2, ..., d_{\text{DIM}-1}, d_{\text{DIM}+1}, ..., d_n)$. BOUNDARY may be omitted for the data types in the following table and, in this case, it is as if it were present with the scalar value shown.

Type of ARRAY	Value of BOUNDARY
Integer	0
Real	0.0
Complex	(0.0, 0.0)
Logical	false
Character (*len*)	*len* blanks

Result Type, Type Parameter, and Shape. The result has the type, type parameters, and shape of ARRAY.

Result Value. Element $(s_1, s_2, ..., s_n)$ of the result has the value ARRAY $(s_1, s_2, ..., s_{DIM-1}, s_{DIM}+sh, s_{DIM+1}, ..., s_n)$ where sh is SHIFT or SHIFT $(s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n)$ provided the inequality LBOUND (ARRAY, DIM) $\le s_{DIM} + sh \le$ UBOUND (ARRAY, DIM) holds and is otherwise BOUNDARY or BOUNDARY $(s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n)$.

Examples.

Case (i): If V is the array (/ 1, 2, 3, 4, 5, 6 /), the effect of shifting V end-off to the left by 3 positions is achieved by EOSHIFT (V, DIM=1, SHIFT=3) which has the value (/ 4, 5, 6, 0, 0, 0 /); EOSHIFT (V, DIM=1, SHIFT=–2, BOUNDARY=99) achieves an end-off shift to the right by 2 positions with the boundary value of 99 and has the value (/ 99, 99, 1, 2, 3, 4 /).

Case (ii): The rows of an array of rank two may all be shifted by the same amount or by different amounts and the boundary elements can be the same or different. If M is the array $\begin{bmatrix} A & B & C \\ A & B & C \\ A & B & C \end{bmatrix}$, then the value of EOSHIFT (M, DIM=2, SHIFT=–1, BOUNDARY='*') is $\begin{bmatrix} * & A & B \\ * & A & B \\ * & A & B \end{bmatrix}$, and the value of EOSHIFT (M, DIM=2, SHIFT=(/ –1, 1, 0 /), BOUNDARY=(/ '*', '/', '?' /)) is $\begin{bmatrix} * & A & B \\ B & C & / \\ A & B & C \end{bmatrix}$.

≡ NEAREST(X,1.0)? SPACING ? be more precise

13.13.33 EPSILON (X).

Description. Returns a positive model number that is almost negligible compared to unity in the model representing numbers of the same type and type parameter as the argument.

Class. Inquiry function.

Argument. X must be of type real. It may be scalar or array valued.

Result Type, Type Parameter, and Shape. Scalar of the same type and type parameter as X.

Result Value. The result has the value b^{1-p} where b and p are as defined in 13.7.1 for the model representing numbers of the same type and type parameter as X.

Example. EPSILON (X) has the value 2^{-23} for real X whose model is as at the end of 13.7.1.

13.13.34 EXP (X).

Description. Exponential.

Class. Elemental function.

Argument. X must be of type real or complex.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to e^X. If X is of type complex, its imaginary part is regarded as a value in radians.

Example. EXP (1.0) has the value 2.7182818 (approximately).

13.13.35 EXPONENT (X).

Description. Returns the exponent part of the argument when represented as a model number.

Class. Elemental function.

Argument. X must be of type real.

Result Type. Default integer.

Result Type and Type Parameters. The result has a value equal to the exponent *e* of the model representation (13.7.1) for the value of X, provided X is nonzero and *e* is within the range for integers. EXPONENT (X) has the value zero if X is zero.

Examples. EXPONENT (1.0) has the value 1 and EXPONENT (4.1) has the value 3 for reals whose model is as at the end of 13.7.1.

13.13.36 FLOOR (A).

Description. Returns the greatest integer less than or equal to its argument.

Class. Elemental function.

Argument. A must be of type real.

Result Type and Type Parameter. Default integer.

Result Value. The result has value equal to the greatest integer less than or equal to A. The result is undefined if the processor cannot represent this value in the default integer type.

Example. FLOOR (3.7) has the value 3. FLOOR (–3.7) has the value –4.

13.13.37 FRACTION (X).

Description. Returns the fractional part of the model representation of the argument value.

Class. Elemental function.

Argument. X must be of type real.

Result Type and Type Parameter. Same as X.

Result Value. The result has the value $X \times b^{-e}$, where *b* and *e* are as defined in 13.7.1 for the model representation of X. If X has the value zero, the result has the value zero.

Example. FRACTION (3.0) has the value 0.75 for reals whose model is as at the end of 13.7.1.

13.13.38 HUGE (X).

Description. Returns the largest number in the model representing numbers of the same type and type parameter as the argument.

Class. Inquiry function.

Argument. X must be of type integer or real. It may be scalar or array valued.

Result Type, Type Parameter, and Shape. Scalar of the same type and type parameter as X.

Result Value. The result has the value r^q-1 if X is of type integer and $(1-b^{-p})b^{e_{max}}$ if X is of type real, where *r*, *q*, *b*, *p*, and e_{max} are as defined in 13.7.1 for the model representing

numbers of the same type and type parameter as X.

Example. HUGE (X) has the value $(1-2^{-24})\times 2^{127}$ for real X whose model is as at the end of 13.7.1.

13.13.39 IACHAR (C).

Description. Returns the position of a character in the ASCII collating sequence.

Class. Elemental function.

Argument. C must be of type character and of length one.

Result Type and Type Parameter. Default integer.

Result Value. If C is in the collating sequence described in ANSI X3.4-1977 (ASCII), the result is the position of C in that sequence and satisfies the inequality ($0 \le$ IACHAR (C) ≤ 127). A processor-dependent value is returned if C is not in the ASCII collating sequence. The results must be consistent with the LGE, LGT, LLE, and LLT lexical comparison functions. For example, if LLE (C, D) is true, IACHAR (C) .LE. IACHAR (D) is true where C and D are any two characters representable by the processor.

Example. IACHAR ('X') has the value 88.

13.13.40 IAND (I, J).

Description. Performs a logical AND.

Class. Elemental function.

Arguments.

I	must be of type integer.
J	must be of type integer with the same kind parameter as I.

Result Type and Type Parameter. Same as I.

Result Value. The result has the value obtained by combining I and J bit-by-bit according to the following truth table:

I	J	IAND (I, J)
1	1	1
1	0	0
0	1	0
0	0	0

Example. IAND (1, 3) has the value 1.

13.13.41 IBCLR (I, POS).

Description. Clears one bit to zero.

Class. Elemental function.

Arguments.

I	must be of type integer.
POS	must be of type integer. It must be nonnegative and less than BIT_SIZE (I).

Result Type and Type Parameter. Same as I.

Result Value. The result has the value of the sequence of bits of I, except that bit POS of I is set to zero.

Example. IBCLR (14, 1) has the result 12.

13.13.42 IBITS (I, POS, LEN).

Description. Extracts a sequence of bits.

Class. Elemental function.

Arguments.

I	must be of type integer.
POS	must be of type integer. It must be nonnegative and POS + LEN must be less than or equal to BIT_SIZE (I).
LEN	must be of type integer and positive.

Result Type and Type Parameter. Same as I.

Result Value. The result has the value of the sequence of LEN bits in I beginning at bit POS right-adjusted and with all other bits zero.

Example. IBITS (14, I, 3) has the value 7.

13.13.43 IBSET (I, POS).

Description. Sets one bit to one.

Class. Elemental function.

Arguments.

I	must be of type integer.
POS	must be of type integer. It must be nonnegative and less than BIT_SIZE (I).

Result Type and Type Parameter. Same as I.

Result Value. The result has the value of the sequence of bits of I, except that bit POS of I is set to one.

Example. IBSET (12, 1) has the value 14.

13.13.44 ICHAR (C).

Description. Returns the position of a character in the processor collating sequence.

Class. Elemental function.

Argument. C must be of type character and of length one. Its value must be that of a character capable of representation in the processor.

Result Type and Type Parameter. Default integer.

Result Value. The result is the position of C in the processor collating sequence and is in the range $0 \leq$ ICHAR (C) $\leq n-1$, where n is the number of characters in the collating sequence. For any characters C and D capable of representation in the processor, C .LE. D is true if and only if ICHAR (C) .LE. ICHAR (D) is true and C .EQ. D is true if and only if ICHAR (C). EQ. ICHAR (D) is true.

Example. ICHAR ('X') has the value 88 on a processor using the ASCII collating sequence.

13.13.45 IEOR (I, J).

Description. Performs an exclusive OR.

Class. Elemental function.

Arguments.

I	must be of type integer.
J	must be of type integer with the same kind type parameter as I.

Result Type and Type Parameter. Same as I.

Result Value. The result has the value obtained by combining I and J bit-by-bit according to the following truth table:

I	J	BIT_XOR (I, J)
1	1	0
1	0	1
0	1	1
0	0	0

Example. IEOR (1, 3) has the value 2.

13.13.46 INDEX (STRING, SUBSTRING, BACK).

Optional Argument. BACK

Description. Returns the starting position of a substring within a string.

Class. Elemental function.

Arguments.

STRING	must be of type character.
SUBSTRING	must be of type character with the same kind type parameter as STRING.
BACK (optional)	must be of type logical.

Result Type and Type Parameter. Default integer.

Result Value.

Case (i): If BACK is absent or present with the value .FALSE., the result is the minimum value of I such that STRING (I : I + LEN (SUBSTRING) – 1) == SUBSTRING or zero if there is no such value. Zero is returned if LEN (STRING) < LEN (SUBSTRING) and one is returned if LEN (SUBSTRING) = 0.

Case (ii): If BACK is present with the value .TRUE., the result is the maximum value of I such that STRING (I : I + LEN (SUBSTRING) – 1) == SUBSTRING or zero if there is no such value. Zero is returned if LEN (STRING) < LEN (SUBSTRING) and LEN (STRING) + 1 is returned if LEN (SUBSTRING) = 0.

Examples. INDEX ('FORTRAN', 'R') has the value 3.
INDEX ('FORTRAN', 'R', BACK = .TRUE.) has the value 5.

13.13.47 INT (A, KIND).

Optional Argument. KIND

Description. Convert to integer type.

Class. Elemental function.

Arguments.

A	must be of type integer, real, or complex.
KIND (optional)	must be a scalar integer constant expression.

Result Type and Type Parameter. Integer. If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of default integer type.

Result Value.

Case (i): If A is of type integer, INT (A) = A.

Case (ii): If A is of type real, there are two cases: if $|A| < 1$, INT (A) has the value 0; if $|A| \geq 1$, INT (A) is the integer whose magnitude is the largest integer that does not exceed the magnitude of A and whose sign is the same as the sign of A.

Case (iii): If A is of type complex, INT (A) is the value obtained by applying the case (ii) rule to the real part of A.

Example. INT (–3.7) has the value –3.

13.13.48 IOR (I, J).

Description. Performs an inclusive OR.

Class. Elemental function.

Arguments.

I	must be of type integer.
J	must be of type integer with the same kind type parameter as I.

Result Type and Type Parameter. Same as I.

Result Value. The result has the value obtained by combining I and J bit-by-bit according to the following truth table:

I	J	IOR (I, J)
1	1	1
1	0	1
0	1	1
0	0	0

Example. IOR (1, 3) has the value 3.

13.13.49 ISHFT (I, SHIFT).

Description. Performs a logical shift.

Class. Elemental function.

Arguments.

I	must be of type integer.
SHIFT	must be of type integer. The absolute value of SHIFT must be less than BIT_SIZE (I).

Result Type and Type Parameter. Same as I.

Result Value. The result has the value obtained by shifting the bits of I by SHIFT positions. If SHIFT is positive, the shift is to the left; if SHIFT is negative, the shift is to the right; and if SHIFT is zero, no shift is performed. Bits shifted out from the left or from the right, as appropriate, are lost. Zeros are shifted in from the opposite end.

Example. ISHFT (3, 1) has the result 6.

13.13.50 ISHFTC (I, SHIFT, SIZE).

Optional Argument. SIZE

Description. Performs a circular shift of the rightmost bits.

Class. Elemental function.

Arguments.

I	must be of type integer.
SHIFT	must be of type integer. The absolute value of SHIFT must be less than or equal to SIZE.
SIZE (optional)	must be of type integer. The value of SIZE must be positive and must not exceed BIT_SIZE (I). If SIZE is absent, it is as if it were present with the value of BIT_SIZE (I).

Result Type and Type Parameter. Same as I.

Result Value. The result has the value obtained by shifting the SIZE rightmost bits of I circularly by SHIFT positions. If SHIFT is positive, the shift is to the left; if SHIFT is negative, the shift is to the right; and if SHIFT is zero, no shift is performed. No bits are lost. The unshifted bits are unaltered.

Example. ISHFTC (3, 2, 3) has the value 5.

13.13.51 KIND (X).

Description. Returns the value of the kind type parameter of X.

Class. Inquiry function.

Argument. X may be of any type with a type parameter named KIND.

Result Type, Type Parameter, and Shape. Default integer scalar.

Result Value. The result has a value equal to the kind type parameter value of X.

Example. KIND (0.0) has the kind type parameter value of default real.

13.13.52 LBOUND (ARRAY, DIM).

Optional Argument. DIM

Description. Returns all the lower bounds of an array or a specified lower bound.

Class. Inquiry function.

Arguments.

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.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \leq DIM \leq n$, where n is the rank of ARRAY.

Result Type, Type Parameter, and Shape. The result is of type default integer. It is scalar if DIM is present; otherwise, the result is an array of rank one and size n, where n is the rank of ARRAY.

Result Value.

Case (i): LBOUND (ARRAY, DIM) has a value equal to the lower bound for subscript DIM of ARRAY if dimension DIM of ARRAY does not have size zero and has the value 1 if dimension DIM has size zero. For an array section or an array expression, it has the value 1.

Case (ii): LBOUND (ARRAY) has a value whose *i*th component is equal to LBOUND (ARRAY, *i*), for *i* = 1, 2, ..., *n*, where *n* is the rank of ARRAY.

Example. If A is declared by the statement

```
REAL A (2:3, 7:10)
```

then LBOUND (A) is (/ 2, 7 /) and LBOUND (A, DIM=2) is 7.

13.13.53 LEN (STRING).

Description. Returns the length of a character entity.

Class. Inquiry function.

Argument. STRING must be of type character. It may be scalar or array valued.

Result Type, Type Parameter, and Shape. Default integer scalar.

Result Value. The result has a value equal to the number of characters in STRING if it is scalar or in an element of STRING if it is array valued.

Example. If C is declared by the statement

```
CHARACTER (11) C (100)
```

LEN (C) has the value 11.

13.13.54 LEN_TRIM (STRING).

Description. Returns the length of the character argument without counting trailing blank characters.

Class. Elemental function.

Argument. STRING must be of type character.

Result Type and Type Parameter. Default integer.

Result Value. The result has a value equal to the number of characters after any trailing blanks in STRING are removed. If the argument contains no nonblank characters, the result is zero.

Examples. LEN_TRIM (' A B ') has the value 4 and LEN_TRIM (' ') has the value 0.

13.13.55 LGE (STRING_A, STRING_B).

Description. Test whether a string is lexically greater than or equal to another string, based on the ASCII collating sequence.

Class. Elemental function.

Arguments.

STRING_A	must be of type character.
STRING_B	must be of type character with the same kind type parameter value as STRING_A.

Result Type and Type Parameters. Default logical.

Result Value. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string. If either string contains a character not in the ASCII character set, the result is processor dependent. The result is true if the strings are equal or if STRING_A follows STRING_B in the collating sequence described in ANSI X3.4-1977 (ASCII); otherwise, the result is false.

Example. LGE ('ONE', 'TWO') has the value .FALSE.

13.13.56 LGT (STRING_A, STRING_B).

Description. Test whether a string is lexically greater than another string, based on the ASCII collating sequence.

Class. Elemental function.

Arguments.

STRING_A	must be of type character.
STRING_B	must be of type character with the same kind type parameter value as STRING_A.

Result Type and Type Parameters. Default logical.

Result Value. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string. If either string contains a character not in the ASCII character set, the result is processor dependent. The result is true if STRING_A follows STRING_B in the collating sequence described in ANSI X3.4-1977 (ASCII); otherwise, the result is false.

Example. LGT ('ONE', 'TWO') has the value .FALSE.

13.13.57 LLE (STRING_A, STRING_B).

Description. Test whether a string is lexically less than or equal to another string, based on the ASCII collating sequence.

Class. Elemental function.

Arguments.

STRING_A	must be of type character.
STRING_B	must be of type character with the same kind type parameter value as STRING_A.

Result Type and Type Parameters. Default logical.

Result Value. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string. If either string contains a character not in the ASCII character set, the result is processor dependent. The result is true if the strings are equal or if STRING_A precedes STRING_B in the collating sequence described in ANSI X3.4-1977 (ASCII); otherwise, the result is false.

Example. LLE ('ONE', 'TWO') has the value .TRUE.

13.13.58 LLT (STRING_A, STRING_B).

Description. Test whether a string is lexically less than another string, based on the ASCII collating sequence.

Class. Elemental function.

Arguments.

STRING_A	must be of type character.
STRING_B	must be of type character with the same kind type parameter value as STRING_A.

Result Type and Type Parameters. Default logical.

Result Value. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string. If either string contains a character not in the ASCII character set, the result is processor dependent. The result is true if STRING_A precedes STRING_B in the collating sequence described in ANSI X3.4-1977 (ASCII); otherwise, the result is false.

Example. LLT ('ONE', 'TWO') has the value .TRUE.

13.13.59 LOG (X).

Description. Natural logarithm.

Class. Elemental function.

Argument. X must be of type real or complex. If X is real, its value must be greater than zero. If X is complex, its value must not be zero.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to $\log_e X$. A result of type complex is the principal value with imaginary part ω in the range $-\pi < \omega \leq \pi$. The imaginary part of the result is π only when the real part of the argument is less than zero and the imaginary part of the argument is zero.

Example. LOG (10.0) has the value 2.3025851 (approximately).

13.13.60 LOG10 (X).

Description. Common logarithm.

Class. Elemental function.

Argument. X must be of type real. The value of X must be greater than zero.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to $\log_{10} X$.

Example. LOG10 (10.0) has the value 1.0 (approximately).

13.13.61 LOGICAL (L, KIND).

Optional Argument. KIND

Description. Converts between kinds of logical.

Class. Elemental function.

Arguments.

L	must be of type logical.
KIND (optional)	must be a scalar integer constant expression.

Result Type and Type Parameter. Logical. If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of default logical.

Result Value.

Case (i): The value is that of L.

Example. LOGICAL (L .OR. .NOT. L) has the value true and is of type default logical, regardless of the kind type parameter of the logical variable L.

13.13.62 MATMUL (MATRIX_A, MATRIX_B).

Description. Performs matrix multiplication of numeric or logical matrices.

Class. Transformational function.

Arguments.

MATRIX_A	must be of numeric type (integer, real, or complex) or of logical type. It must be array valued and of rank one or two.

MATRIX_B must be of numeric type if MATRIX_A is of numeric type and of logical type if MATRIX_A is of logical type. It must be array valued and of rank one or two. If MATRIX_A has rank one, MATRIX_B must have rank two. If MATRIX_B has rank one, MATRIX_A must have rank two. The size of the first (or only) dimension of MATRIX_B must equal the size of the last (or only) dimension of MATRIX_A.

Result Type, Type Parameter, and Shape. If the arguments are of numeric type, the type and type parameter of the result are determined by the types of the arguments according to 7.1.4. If the arguments are of type logical with the same kind type parameter, the result is of type logical. The shape of the result depends on the shapes of the arguments as follows:

Case (i): If MATRIX_A has shape (*n*, *m*) and MATRIX_B has shape (*m*, *k*), the result has shape (*n*, *k*).

Case (ii): If MATRIX_A has shape (*m*) and MATRIX_B has shape (*m*, *k*), the result has shape (*k*).

Case (iii): If MATRIX_A has shape (*n*, *m*) and MATRIX_B has shape (*m*), the result has shape (*n*).

Result Value.

Case (i): Element (*i*, *j*) of the result has the value SUM (MATRIX_A (*i*, :) * MATRIX_B (:, *j*)) if the arguments are of numeric type and has the value ANY (MATRIX_A (*i*, :) .AND. MATRIX_B (:, *j*)) if the arguments are of logical type.

Case (ii): Element (*j*) of the result has the value SUM (MATRIX_A (:) * MATRIX_B (:, *j*)) if the arguments are of numeric type and has the value ANY (MATRIX_A (:) .AND. MATRIX_B (:, *j*)) if the arguments are of logical type.

Case (iii): Element (*i*) of the result has the value SUM (MATRIX_A (*i*, :) * MATRIX_B (:)) if the arguments are of numeric type and has the value ANY (MATRIX_A (*i*, :) .AND. MATRIX_B (:)) if the arguments are of logical type.

Examples. Let A and B be the matrices $\begin{bmatrix}1 & 2 & 3\\ 2 & 3 & 4\end{bmatrix}$ and $\begin{bmatrix}1 & 2\\ 2 & 3\\ 3 & 4\end{bmatrix}$; let X and Y be the vectors (/ 1, 2 /) and (/ 1, 2, 3 /).

Case (i): The result of MATMUL (A, B) is the matrix-matrix product AB with the value $\begin{bmatrix}14 & 20\\ 20 & 29\end{bmatrix}$.

Case (ii): The result of MATMUL (X, A) is the vector-matrix product XA with the value (/ 5, 8, 11 /).

Case (iii): The result of MATMUL (A, Y) is the matrix-vector product AY with the value (/ 14, 20 /).

13.13.63 MAX (A1, A2, A3, ...).

Optional Arguments. A3, ...

Description. Maximum value.

Class. Elemental function.

Arguments. The arguments must all have the same type which must be integer or real and they must all have the same type parameter.

Result Type and Type Parameter. Same as the arguments.

Result Value. The value of the result is that of the largest argument.

Example. MAX (–9.0, 7.0, 2.0) has the value 7.0.

13.13.64 MAXEXPONENT (X).

Description. Returns the maximum exponent in the model representing numbers of the same type and type parameter as the argument.

Class. Inquiry function.

Argument. X must be of type real. It may be scalar or array valued.

Result Type, Type Parameter, and Shape. Default integer scalar.

Result Value. The result has the value e_{max}, as defined in 13.7.1 for the model representing numbers of the same type and type parameter as X.

Example. MAXEXPONENT (X) has the value 127 for real X whose model is as at the end of 13.7.1.

13.13.65 MAXLOC (ARRAY, MASK).

Optional Argument. MASK

Description. Determine the location of an element of ARRAY having the maximum value of the elements identified by MASK.

Class. Transformational function.

Arguments.

ARRAY	must be of type integer or real. It must not be scalar.
MASK (optional)	must be of type logical and must be conformable with ARRAY.

Result Type, Type Parameter, and Shape. The result is of type default integer; it is an array of rank one and of size equal to the rank of ARRAY.

Result Value.

Case (i): If MASK is absent, the result is a rank-one array whose element values are the values of the subscripts of an element of ARRAY whose value equals the maximum value of all of the elements of ARRAY. The *i*th subscript returned lies in the range 1 to e_i, where e_i is the extent of the *i*th dimension of ARRAY. If more than one element has the maximum value, the element whose subscripts are returned is processor dependent. If ARRAY has size zero, the value of the result is processor dependent.

Case (ii): If MASK is present, the result is a rank-one array whose element values are the values of the subscripts of an element of ARRAY, corresponding to a true element of MASK, whose value equals the maximum value of all such elements of ARRAY. The *i*th subscript returned lies in the range 1 to e_i, where e_i is the extent of the *i*th dimension of ARRAY. If more than one such element has the maximum value, the element whose subscripts are returned is processor dependent. If there are no such elements (that is, if ARRAY has size zero or every element of MASK has the value false), the value of the result is processor dependent.

Examples.

Case (i): The value of MAXLOC ((/ 2, 4, 6 /)) is (/ 3 /).

Case (ii): If A has the value $\begin{bmatrix} 0 & -5 & 8 & -3 \\ 3 & 4 & -1 & 2 \\ 1 & 5 & 6 & -4 \end{bmatrix}$, MAXLOC (A, MASK=A.LT.6) has the value (/ 3, 2 /).

13.13.66 MAXVAL (ARRAY, DIM, MASK).

Optional Arguments. DIM, MASK

Description. Maximum value of the elements of ARRAY along dimension DIM corresponding to the true elements of MASK.

Class. Transformational function.

Arguments.

ARRAY	must be of type integer or real. It must not be scalar.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \le \text{DIM} \le n$, where n is the rank of ARRAY.
MASK (optional)	must be of type logical and must be conformable with ARRAY.

Result Type, Type Parameter, and Shape. The result is of the same type and type parameter as ARRAY. It is scalar if DIM is absent or ARRAY has rank one; otherwise, the result is an array of rank $n-1$ and of shape $(d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of ARRAY.

Result Value.

Case (i): The result of MAXVAL (ARRAY) has a value equal to the maximum value of all the elements of ARRAY or has the value –HUGE (ARRAY) if ARRAY has size zero.

Case (ii): The result of MAXVAL (ARRAY, MASK = MASK) has a value equal to the maximum value of the elements of ARRAY corresponding to true elements of MASK or has the value –HUGE (ARRAY) if there are no true elements.

Case (iii): If ARRAY has rank one, MAXVAL (ARRAY, DIM [,MASK]) has a value equal to that of MAXVAL (ARRAY [,MASK = MASK]). Otherwise, the value of element $(s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n)$ of MAXVAL (ARRAY, DIM [,MASK]) is equal to MAXVAL (ARRAY $(s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n)$, [, MASK = MASK $(s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n)$]).

Examples.

Case (i): The value of MAXVAL ((/ 1, 2, 3 /)) is 3.

Case (ii): MAXVAL (C, MASK = C .LT. 0.0) finds the maximum of the negative elements of C.

Case (iii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$, MAXVAL (B, DIM=1) is (/ 2, 4, 6 /) and MAXVAL (B, DIM=2) is (/ 5, 6 /).

13.13.67 MERGE (TSOURCE, FSOURCE, MASK).

Description. Choose alternative value according to the value of a mask.

Class. Elemental function.

Arguments.

TSOURCE	may be of any type.
FSOURCE	must be of the same type and type parameters as TSOURCE.
MASK	must be of type logical.

Result Type and Type Parameters. Same as TSOURCE.

Result Value. The result is TSOURCE if MASK is true and FSOURCE otherwise.

Example. If TSOURCE is the array $\begin{bmatrix} 1 & 6 & 5 \\ 2 & 4 & 6 \end{bmatrix}$, FSOURCE is the array $\begin{bmatrix} 0 & 3 & 2 \\ 7 & 4 & 8 \end{bmatrix}$ and MASK is

the array $\begin{bmatrix} T & . & T \\ . & . & T \end{bmatrix}$, where "T" represents true and "." represents .FALSE., then MERGE (TSOURCE, FSOURCE, MASK) is $\begin{bmatrix} 1 & 3 & 5 \\ 7 & 4 & 6 \end{bmatrix}$.

13.13.68 MIN (A1, A2, A3, ...).

Optional Arguments. A3, ...

Description. Minimum value.

Class. Elemental function.

Arguments. The arguments must all be of the same type which must be integer or real and they must all have the same type parameter.

Result Type and Type Parameter. Same as the arguments.

Result Value. The value of the result is that of the smallest argument.

Example. MIN (–9.0, 7.0, 2.0) has the value –9.0.

13.13.69 MINEXPONENT (X).

Description. Returns the minimum (most negative) exponent in the model representing numbers of the same type and type parameter as the argument.

Class. Inquiry function.

Argument. X must be of type real. It may be scalar or array valued.

Result Type, Type Parameter, and Shape. Default integer scalar.

Result Value. The result has the value e_{min}, as defined in 13.7.1 for the model representing numbers of the same type and type parameter as X.

Example. MINEXPONENT (X) has the value –126 for real X whose model is as at the end of 13.7.1.

13.13.70 MINLOC (ARRAY, MASK).

Optional Argument. MASK

Description. Determine the location of an element of ARRAY having the minimum value of the elements identified by MASK.

Class. Transformational function.

Arguments.

ARRAY must be of type integer or real. It must not be scalar.
MASK (optional) must be of type logical and must be conformable with ARRAY.

Result Type, Type Parameter, and Shape. The result is of type default integer; it is an array of rank one and of size equal to the rank of ARRAY.

Result Value.

Case (i): If MASK is absent, the result is a rank-one array whose element values are the values of the subscripts of an element of ARRAY whose value equals the minimum value of all the elements of ARRAY. The *i*th subscript returned lies in the range 1 to e_i, where e_i is the extent of the *i*th dimension of ARRAY. If more than one element has the minimum value, the element whose subscripts are returned is processor dependent. If ARRAY has size zero, the value of the result is processor dependent.

Case (ii): If MASK is present, the result is a rank-one array whose element values are the values of the subscripts of an element of ARRAY, corresponding to a true element of MASK, whose value equals the minimum value of all such elements of ARRAY. The ith subscript returned lies in the range 1 to e_i, where e_i is the extent of the ith dimension of ARRAY. If more than one such element has the minimum value, the element whose subscripts are returned is processor dependent. If ARRAY has size zero or every element of MASK has the value .FALSE., the value of the result is processor dependent.

Examples.

Case (i): The value of MINLOC ((/ 2, 4, 6 /)) is (/ 1 /).

Case (ii): If A has the value $\begin{bmatrix} 0 & -5 & 8 & -3 \\ 3 & 4 & -1 & 2 \\ 1 & 5 & 6 & -4 \end{bmatrix}$, MINLOC (A, MASK=A.GT.–4) has the value (/ 1, 4 /).

13.13.71 MINVAL (ARRAY, DIM, MASK).

Optional Arguments. DIM, MASK

Description. Minimum value of all the elements of ARRAY along dimension DIM corresponding to true elements of MASK.

Class. Transformational function.

Arguments.

ARRAY	must be of type integer or real. It must not be scalar.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \le \text{DIM} \le n$, where n is the rank of ARRAY.
MASK (optional)	must be of type logical and must be conformable with ARRAY.

Result Type, Type Parameter, and Shape. The result is of the same type and type parameter as ARRAY. It is scalar if DIM is absent or ARRAY has rank one; otherwise, the result is an array of rank $n-1$ and of shape $(d_1, d_2, ..., d_{\text{DIM}-1}, d_{\text{DIM}+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of ARRAY.

Result Value.

Case (i): The result of MINVAL (ARRAY) has a value equal to the minimum the value of all the elements of ARRAY or has the value HUGE (ARRAY) if ARRAY has size zero.

Case (ii): The result of MINVAL (ARRAY, MASK = MASK) has a value equal to the minimum value of the elements of ARRAY corresponding to true elements of MASK or has the value HUGE (ARRAY) if there are no true elements.

Case (iii): If ARRAY has rank one, MINVAL (ARRAY, DIM [,MASK]) has a value equal to that of MINVAL (ARRAY [,MASK = MASK]). Otherwise, the value of element $(s_1, s_2, ..., s_{\text{DIM}-1}, s_{\text{DIM}+1}, ..., s_n)$ of MINVAL (ARRAY, DIM [,MASK]) is equal to MINVAL (ARRAY $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$ [, MASK= MASK $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$]).

Examples.

Case (i): The value of MINVAL ((/ 1, 2, 3 /)) is 1.

Case (ii): MINVAL (C, MASK = C .GT. 0.0) forms the minimum of the positive elements of C.

Case (iii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$, MINVAL (B, DIM=1) is (/ 1, 3, 5 /) and MINVAL (B, DIM=2) is (/ 1, 2 /).

13.13.72 MOD (A, P).

Description. Remainder function.

Class. Elemental function.

Arguments.

A	must be of type integer or real.
P	must be of the same type and type parameter as A.

Result Type and Type Parameter. Same as A.

Result Value. If P≠0, the value of the result is A–INT (A/P) * P. If P = 0, the result is undefined.

Example. MOD (3.0, 2.0) has the value 1.0.

13.13.73 MODULO (A, P).

Description. Modulo function.

Class. Elemental function.

Arguments.

A	must be of type integer or real.
P	must be of the same type and kind type parameter as A.

Result Type and Type Parameter. Same as A.

Result Value.

Case (i): A is of type integer. If P ≠ 0, the value of the result is A - FLOOR (REAL (A) / REAL (P)) * P. If P = 0, the result is undefined.

Case (ii): A is of type real. If P ≠ 0, the value of the result is A - FLOOR (A / P) * P. If P = 0, the result is undefined.

Examples. MODULO (8, 5) has the value 3. MODULO (–8, 5) has the value 2. MODULO (8, –5) has the value –2. MODULO (–8, –5) has the value –3.

13.13.74 MVBITS (FROM, FROMPOS, LEN, TO, TOPOS).

Description. Copies a sequence of bits from one data object to another.

Class. Elemental subroutine.

Arguments.

FROM	must be of type integer.
FROMPOS	must be of type integer. FROMPOS + LEN must be less than or equal to BIT_SIZE (FROM).
LEN	must be of type integer and positive. It must be conformable with TO.
TO	must be a variable of type integer with the same kind type parameter value as FROM and may be the same variable as FROM. TO is set by copying the sequence of bits of length LEN, starting at positions FROMPOS of FROM to positions TOPOS of TO. No other bits of TO are altered. On return, the LEN bits of TO starting at TOPOS are equal to the value that the LEN bits of FROM starting at FROMPOS had on entry.
TOPOS	must be of type integer and nonnegative. If TO is a scalar, TOPOS + LEN must be less than or equal to BIT_SIZE (TO).

Example. If TO has the initial value 6, the value of TO after the statement CALL MVBITS (7, 2, 2, TO, 0) is 5.

13.13.75 NEAREST (X, S).

Description. Returns the nearest different machine representable number in a given direction.

Class. Elemental function.

Arguments.

X	must be of type real.
S	must be of type real and not equal to zero.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to the machine representable number distinct from X and nearest to it in the direction of the infinity with the same sign as S.

Example. NEAREST (3.0, 2.0) has the value $3+2^{-22}$ on a machine whose representation is that of the model at the end of 13.7.1.

13.13.76 NINT (A, KIND).

Optional Argument. KIND

Description. Nearest integer.

Class. Elemental function.

Arguments.

A	must be of type real.
KIND (optional)	must be a scalar integer constant expression.

Result Type and Type Parameter. Integer. If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of default integer type.

Result Value. If A > 0, NINT (A) has the value INT (A+0.5); if A≤0, NINT (A) has the value INT (A–0.5).

Example. NINT (2.783) has the value 3.

13.13.77 NOT (I).

Description. Performs a logical complement.

Class. Elemental function.

Argument. I must be of type integer.

Result Type and Type Parameter. Same as I.

Result Value. The result has the value obtained by complementing I bit-by-bit according to the following truth table:

I	NOT (I)
1	0
0	1

Example. If I is represented by the string of bits 01010101, NOT (I) has the binary value 10101010.

13.13.78 PACK (ARRAY, MASK, VECTOR).

Optional Argument. VECTOR

Description. Pack an array into an array of rank one under the control of a mask.

Class. Transformational function.

Arguments.

ARRAY	may be of any type. It must not be scalar.
MASK	must be of type logical and must be conformable with ARRAY.
VECTOR (optional)	must be of the same type and type parameters as ARRAY and must have rank one. VECTOR must have at least as many elements as there are true elements in MASK. If MASK is scalar with the value true, VECTOR must have at least as many elements as there are in ARRAY.

Result Type, Type Parameter, and Shape. The result is an array of rank one with the same type and type parameters as ARRAY. If VECTOR is present, the result size is that of VECTOR; otherwise, the result size is the number t of true elements in MASK unless MASK is scalar with the value true, in which case the result size is the size of ARRAY.

Result Value. Element i of the result is the element of ARRAY that corresponds to the ith true element of MASK, taking elements in array element order, for $i = 1, 2, ..., t$. If VECTOR is present and has size $n > t$, element i of the result has the value VECTOR (i), for $i = t+1, ..., n$.

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 the function PACK. The result of PACK (M, MASK = M .NE. 0) is (/ 9, 7 /) and the result of PACK (M, M .NE. 0, VECTOR = (/ 2, 4, 6, 8, 10, 12 /)) is (/ 9, 7, 6, 8, 10, 12 /).

13.13.79 PRECISION (X).

Description. Returns the decimal precision in the model representing real numbers with the same kind type parameter as the argument.

Class. Inquiry function.

Argument. X must be of type real or complex. It may be scalar or array valued.

Result Type, Type Parameter, and Shape. Default integer scalar.

Result Value. The result has the value INT ((*p*−1) * LOG10 (*b*)) + *k*, where *b* and *p* are as defined in 13.7.1 for the model representing real numbers with the same value for the kind type parameter as X, and where *k* is 1 if *b* is an integral power of 10 and 0 otherwise.

Example. PRECISION (X) has the value INT (23 * LOG10 (2.)) = INT (6.92...) = 6 for real X whose model is as at the end of 13.7.1.

13.13.80 PRESENT (A).

Description. Determine whether an optional argument is present.

Class. Inquiry function.

Argument. A must be an optional argument of the procedure in which the PRESENT function reference appears.

Result Type and Type Parameters. Default logical scalar.

Result Value. The result has the value .TRUE. if A is present (12.5.2.8) and otherwise has the value .FALSE.

13.13.81 PRODUCT (ARRAY, DIM, MASK).

Optional Arguments. DIM, MASK

Description. Product of all the elements of ARRAY along dimension DIM corresponding to the true elements of MASK.

Class. Transformational function.

Arguments.

ARRAY	must be of type integer, real, or complex. It must not be scalar.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \le \text{DIM} \le n$, where n is the rank of ARRAY.
MASK (optional)	must be of type logical and must be conformable with ARRAY.

Result Type, Type 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 an array of rank $n-1$ and of shape $(d_1, d_2, ..., d_{\text{DIM}-1}, d_{\text{DIM}+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of ARRAY.

Result Value.

Case (i): The result of PRODUCT (ARRAY) has a value equal to a processor-dependent approximation to the product of all the elements of ARRAY or has the value one if ARRAY has size zero.

Case (ii): The result of PRODUCT (ARRAY, MASK = MASK) has a value equal to a processor-dependent approximation to the product of the elements of ARRAY corresponding to the true elements of MASK or has the value one if there are no true elements.

Case (iii): If ARRAY has rank one, PRODUCT (ARRAY, DIM [,MASK]) has a value equal to that of PRODUCT (ARRAY [,MASK = MASK]). Otherwise, the value of element $(s_1, s_2, ..., s_{\text{DIM}-1}, s_{\text{DIM}+1}, ..., s_n)$ of PRODUCT (ARRAY, DIM [,MASK]) is equal to PRODUCT (ARRAY $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$ [, MASK = MASK $(s_1, s_2, ..., s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, ..., s_n)$]).

Examples.

Case (i): The value of PRODUCT ((/ 1, 2, 3 /)) is 6.

Case (ii): PRODUCT (C, MASK = C .GT. 0.0) forms the product of the positive elements of C.

Case (iii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$, PRODUCT (B, DIM=1) is (/ 2, 12, 30 /) and PRODUCT (B, DIM=2) is (/ 15, 48 /).

13.13.82 RADIX (X).

Description. Returns the base of the model representing numbers of the same type and type parameter as the argument.

Class. Inquiry function.

Argument. X must be of type integer or real. It may be scalar or array valued.

Result Type, Type Parameter, and Shape. Default integer scalar.

Result Value. The result has the value r if X is of type integer and the value b if X is of type real, where r and b are as defined in 13.7.1 for the model representing numbers of the same type and type parameter as X.

Example. RADIX (X) has the value 2 for real X whose model is as at the end of 13.7.1.

13.13.83 RANDOM (HARVEST).

Description. Returns one pseudorandom number or an array of pseudorandom numbers from the uniform distribution over the range $0 \le x < 1$.

Class. Subroutine.

Argument. HARVEST must be of type real. It may be a scalar or an array variable. It is set to contain pseudorandom numbers from the uniform distribution in the interval $0 \le x < 1$.

Examples.

```
REAL X, Y (10, 10)
CALL RANDOM (HARVEST = X)  ! INITIALIZES X WITH A PSEUDORANDOM NUMBER
CALL RANDOM (Y)
! X AND Y CONTAIN UNIFORMLY DISTRIBUTED RANDOM NUMBERS
```

13.13.84 RANDOMSEED (SIZE, PUT, GET).

Optional Arguments. SIZE, PUT, GET

Description. Initializes or restarts the pseudorandom number generator.

Class. Subroutine.

Arguments. There must either be exactly one or no arguments present.

SIZE (optional) must be scalar and of type integer. It is set to the number *N* of integers that the processor uses to hold the value of the seed.

PUT (optional) must be an integer array of rank one and size ≥ *N*. It is used by the processor to set the seed value.

GET (optional) must be an integer array of rank one and size ≥ *N*. It is set by the processor to the current value of the seed.

If no argument is present, the processor sets the seed to a processor-dependent value.

Examples.

```
CALL RANDOMSEED  ! PROCESSOR INITIALIZATION
CALL RANDOMSEED (SIZE = K)  ! SETS K = N
CALL RANDOMSEED (PUT = SEED (1 : K))  ! SET USER SEED
CALL RANDOMSEED (GET = OLD (1 : K))  ! READ CURRENT SEED
```

13.13.85 RANGE (X).

Description. Returns the decimal exponent range in the model representing integer or real numbers with the same kind type parameter as the argument.

Class. Inquiry function.

Argument. X must be of type integer, real, or complex. It may be scalar or array valued.

Result Type, Type Parameter, and Shape. Default integer scalar.

Result Value.

Case (i): For an integer argument, the result has the value INT (LOG10 (*huge*)), where *huge* is the largest positive integer in the model representing integer numbers with same kind type parameter as X (13.7.1).

Case (ii): For a real or complex argument, the result has the value INT (MIN (LOG10 (*huge*), –LOG10 (*tiny*))), where *huge* and *tiny* are the largest and smallest positive numbers in the model representing real numbers with the same value for the kind type parameter as X (13.7.1).

Example. RANGE (X) has the value 38 for real X whose model is as at the end of 13.7.1, since in this case *huge* = $(1-2^{-24}) \times 2^{127}$ and *tiny* = 2^{-127}.

13.13.86 REAL (A, KIND).

Optional Argument. KIND

Description. Convert to real type.

Class. Elemental function.

Arguments.

A	must be of type integer, real, or complex.
KIND (optional)	must be a scalar integer constant expression.

Result Type and Type Parameter. Real.

Case (i): If A is of type integer or real and KIND is present, the type parameter is that specified by KIND. If A is of type integer or real and KIND is not present, the type parameter is the processor-dependent type parameter for the default real type.

Case (ii): If A is of type complex and KIND is present, the type parameter is that specified by KIND. If A is of type complex and KIND is not present, the type parameter is the type parameter of A.

Result Value.

Case (i): If A is of type integer or real, the result is equal to a processor-dependent approximation to A. If A is of type double precision real, the result is equal to DBLE (A).

Case (ii): If A is of type complex, the result is equal to a processor-dependent approximation to the real part of A.

Examples. REAL (–3) has the value –3.0. REAL (Z, KIND (Z)) has the same kind type parameter and the same value as the real part of the complex variable Z.

13.13.87 REPEAT (STRING, NCOPIES).

Description. Concatenate several copies of a string.

Class. Transformational function.

Arguments.

STRING	must be scalar and of type character.
NCOPIES	must be scalar and of type integer. Its value must not be negative.

Result Type, Type Parameter, and Shape. Character scalar of length NCOPIES times that of STRING, with the same kind type parameter as STRING.

Result Value. The value of the result is the concatenation of NCOPIES copies of STRING.

Example. REPEAT ('H', 2) has the value 'HH'.

13.13.88 RESHAPE (SHAPE, SOURCE, PAD, ORDER).

Optional Arguments. PAD, ORDER

Description. Change the shape of an array.

Class. Transformational function.

Arguments.

SHAPE	must be of type integer, rank one, and constant size. Its size must be positive and less than 8. It must not have an element whose value is negative.

SOURCE may be of any type. It must be array valued. If PAD is absent or of size zero, the size of SOURCE must be ≥ PRODUCT (SHAPE). The size of the result is the product of the values of the elements of SHAPE.

PAD (optional) must be of the same type and type parameters as SOURCE. PAD must be array valued.

ORDER (optional) must be of type integer, must have the same shape as SHAPE, and its value must be a permutation of (1, 2, ..., *n*), where *n* is the size of SHAPE. If absent, it is as if it were present with value (1, 2, ..., *n*).

Result Type, Type Parameter, and Shape. The result is an array of shape SHAPE (i.e., SHAPE (RESHAPE (SHAPE, SOURCE, PAD, ORDER)) is equal to SHAPE) with the same type and type parameters as SOURCE.

Result Value. The elements of the result, taken in permuted subscript order ORDER (1), ..., ORDER (*n*), are those of SOURCE in normal array element order followed if necessary by those of PAD in array element order, followed if necessary by additional copies of PAD in array element order.

Examples. RESHAPE ((/ 2, 3 /), (/ 1, 2, 3, 4, 5, 6 /)) has the value $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$. RESHAPE ((/ 2, 4 /), (/ 1, 2, 3, 4, 5, 6 /), (/ 0, 0 /), (/ 2, 1 /)) has the value $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 0 & 0 \end{bmatrix}$.

13.13.89 RRSPACING (X).

Description. Returns the reciprocal of the relative spacing of model numbers near the argument value.

Class. Elemental function.

Argument. X must be of type real.

Result Type and Type Parameter. Same as X.

Result Value. The result has the value $|X \times b^{-e}| \times b^{p}$, where *b*, *e*, and *p* are as defined in 13.7.1 for the model representation of X, provided this result is within range.

Example. RRSPACING (–3.0) has the value 0.75×2^{24} for reals whose model is as at the end of 13.7.1.

13.13.90 SCALE (X, I).

Description. Returns $X \times b^{I}$ where *b* is the base in the model representation of X.

Class. Elemental function.

Arguments.

X must be of type real.

I must be of type integer.

Result Type and Type Parameter. Same as X.

Result Value. The result has the value $X \times b^{I}$, where *b* is defined in 13.7.1 for model numbers representing values of X, provided this result is within range.

Example. SCALE (3.0, 2) has the value 12.0 for reals whose model is as at the end of 13.7.1.

13.13.91 SCAN (STRING, SET, BACK).

Optional Argument. BACK

Description. Scan a string for a character in a set of characters.

Class. Elemental function.

Arguments.

STRING	must be of type character.
SET	must be of type character with the same kind type parameter as STRING.
BACK (optional)	must be of type logical.

Result Type and Type Parameter. Default integer.

Result Value. The value of the result is zero if no character of STRING is in SET or if the length of STRING or SET is zero. Otherwise:

Case (i): If BACK is absent or is present with the value .FALSE., the value of the result is the position of the leftmost character of STRING that is in SET.

Case (ii): If BACK is present with the value .TRUE., the value of the result is the position of the rightmost character of STRING that is in SET.

Examples. SCAN ('FORTRAN', 'BCD') has the value 0.

Case (i): SCAN ('FORTRAN', 'TR') has the value 3.

Case (ii): SCAN ('FORTRAN', 'TR', BACK = .TRUE.) has the value 5.

13.13.92 SELECTED_INT_KIND (R).

Description. Returns the smallest value of the kind type parameter of an integer data type that represents all integer values n with $-10^R < n < 10^R$.

Class. Inquiry function.

Arguments.

R	must be scalar and of type integer.

Result Type, Type Parameter, and Shape. Default integer scalar.

Result Value. The result has a value equal to the smallest value of the kind type parameter of an integer data type that represents all values n in the range of values n with $-10^R < n < 10^R$.

Example. SELECTED_INT_KIND (6) has the value KIND (0) on a machine that supports default integer representation method with $r = 2$ and $q = 31$.

13.13.93 SELECTED_REAL_KIND (P, R).

Description. Returns the smallest value of the kind type parameter of a real data type with decimal precision of at least P digits and a decimal exponent range of at least R.

Class. Inquiry function.

Arguments. At least one argument must be present.

P (optional)	must be scalar and of type integer.
R (optional)	must be scalar and of type integer.

Result Type, Type Parameter, and Shape. Default integer scalar.

Result Value. The result has a value equal to the smallest value of the kind type parameter of a real data type with decimal precision of at least P digits and a decimal exponent range

of at least R.

Example. SELECTED_REAL_KIND (6, 70) has the value KIND (0.0) on a machine that supports default real approximation method with $b = 16$, $p = 5$, $e_{min} = 64$, and $e_{max} = 63$.

13.13.94 SETEXPONENT (X, I).

Description. Returns the model number whose fractional part is the fractional part of the model representation of X and whose exponent part is I.

Class. Elemental function.

Arguments.

X	must be of type real.
I	must be of type integer.

Result Type and Type Parameter. Same as X.

Result Value. The result has the value $X \times b^{I-e}$, where b and e are as defined in 13.7.1 for the model representation of X, provided this result is within range. If X has value zero, the result has value zero.

Example. SETEXPONENT (3.0, 1) has the value 1.5 for reals whose model is as at the end of 13.7.1.

13.13.95 SHAPE (SOURCE).

Description. Returns the shape of an array or a scalar.

Class. Inquiry function.

Argument. SOURCE may be of any type. It may be array valued or scalar. It must not be a pointer that is disassociated or an allocatable array that is not allocated. It must not be an assumed-size array.

Result Type, Type Parameter, and Shape. The result is a default integer array of rank one whose size is equal to the rank of SOURCE.

Result Value. The value of the result is the shape of SOURCE.

Examples. The value of SHAPE (A (2:5, –1:1)) is (/ 4, 3 /). The value of SHAPE (3) is the rank-one array of size zero.

13.13.96 SIGN (A, B).

Description. Absolute value of A ~~times~~ the sign of B.

Class. Elemental function.

Arguments.

A	must be of type integer or real.
B	must be of the same type and type parameter as A.

Result Type and Type Parameter. Same as A.

Result Value. The value of the result is |A| if $B \geq 0$ and –|A| if $B < 0$.

Example. SIGN (–3.0, 2.0) has the value 3.0.

13.13.97 SIN (X).

Description. Sine function.

Class. Elemental function.

Argument. X must be of type real or complex.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to sin(X). If X is of type real, it is regarded as a value in radians. If X is of type complex, its real part is regarded as a value in radians.

Example. SIN (1.0) has the value 0.84147098 (approximately).

13.13.98 SINH (X).

Description. Hyperbolic sine function.

Class. Elemental function.

Argument. X must be of type real.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to sinh(X).

Example. SINH (1.0) has the value 1.1752012 (approximately).

13.13.99 SIZE (ARRAY, DIM).

Optional Argument. DIM

Description. Returns the extent of an array along a specified dimension or the total number of elements in the array.

Class. Inquiry function.

Arguments.

ARRAY	may be of any type. It 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.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \le \text{DIM} \le n$, where n is the rank of ARRAY.

Result Type, Type Parameter, and Shape. Default integer scalar.

Result Value. The result has a value equal to the extent of dimension DIM of ARRAY or, if DIM is absent, the total number of elements of ARRAY.

Examples. The value of SIZE (A (2:5, –1:1), DIM=2) is 3. The value of SIZE (A (2:5, –1:1)) is 12.

13.13.100 SPACING (X).

Description. Returns the absolute spacing of model numbers near the argument value.

Class. Elemental function.

Argument. X must be of type real.

Result Type and Type Parameter. Same as X.

Result Value. The result has the value b^{e-p}, where b, e, and p are as defined in 13.7.1 for the model representation of X, provided this result is within range; otherwise, the result is the same as that of TINY (X).

Example. SPACING (3.0) has the value 2^{-22} for reals whose model is as at the end of 13.7.1.

13.13.101 SPREAD (SOURCE, DIM, NCOPIES).

Description. Replicates an array by adding a dimension. Broadcasts several copies of SOURCE along a specified dimension (as in forming a book from copies of a single page) and thus forms an array of rank one greater.

Class. Transformational function.

Arguments.

SOURCE	may be of any type. It may be scalar or array valued. The rank of SOURCE must be less than 7.
DIM	must be scalar and of type integer with value in the range $1 \le DIM \le n+1$, where n is the rank of SOURCE.
NCOPIES	must be scalar and of type integer.

Result Type, Type Parameter, and Shape. The result is an array of the same type and type parameters as SOURCE and of rank $n+1$, where n is the rank of SOURCE.

Case (i): If SOURCE is scalar, the shape of the result is (MAX (NCOPIES, 0)).

Case (ii): If SOURCE is array valued with shape $(d_1, d_2, ..., d_n)$, the shape of the result is $(d_1, d_2, ..., d_{DIM-1}, MAX (NCOPIES, 0), d_{DIM}, ..., d_n)$.

Result Value.

Case (i): If SOURCE is scalar, each element of the result has a value equal to SOURCE.

Case (ii): If SOURCE is array valued, the element of the result with subscripts $(r_1, r_2, ..., r_{n+1})$ has the value SOURCE $(r_1, r_2, ..., r_{DIM-1}, r_{DIM+1}, ..., r_{n+1})$.

Example. If A is the array (/ 2, 3, 4 /), SPREAD (A, DIM=1, NCOPIES=3) is the array $\begin{bmatrix} 2 & 3 & 4 \\ 2 & 3 & 4 \\ 2 & 3 & 4 \end{bmatrix}$.

13.13.102 SQRT (X).

Description. Square root.

Class. Elemental function.

Argument. X must be of type real or complex. Unless X is complex, its value must be greater than or equal to zero.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to the square root of X. A result of type complex is the principal value with the real part greater than or equal to zero. When the real part of the result is zero, the imaginary part is greater than or equal to zero.

Example. SQRT (4.0) has the value 2.0 (approximately).

13.13.103 SUM (ARRAY, DIM, MASK).

Optional Arguments. DIM, MASK

Description. Sum all the elements of ARRAY along dimension DIM with mask MASK.

Class. Transformational function.

Arguments.

ARRAY	must be of type integer, real, or complex. It must not be scalar.

DIM (optional) must be scalar and of type integer with a value in the range $1 \le DIM \le n$, where n is the rank of ARRAY.

MASK (optional) must be of type logical and must be conformable with ARRAY.

Result Type, Type Parameter, and Shape. The result is of the same type and type parameter as ARRAY. It is scalar if DIM is absent or ARRAY has rank one; otherwise, the result is an array of rank $n-1$ and of shape $(d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of ARRAY.

Result Value.

Case (i): The result of SUM (ARRAY) has a value equal to a processor-dependent approximation to the sum of all the elements of ARRAY or has the value zero if ARRAY has size zero.

Case (ii): The result of SUM (ARRAY, MASK = MASK) has a value equal to a processor-dependent approximation to the sum of the elements of ARRAY corresponding to the true elements of MASK or has the value zero if there are no true elements.

Case (iii): If ARRAY has rank one, SUM (ARRAY, DIM [,MASK]) has a value equal to that of SUM (ARRAY [,MASK = MASK]). Otherwise, the value of element $(s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n)$ of SUM (ARRAY, DIM [,MASK]) is equal to SUM (ARRAY $(s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n)$ [, MASK= MASK $(s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n)$]).

Examples.

Case (i): The value of SUM ((/ 1, 2, 3 /)) is 6.

Case (ii): SUM (C, MASK= C .GT. 0.0) forms the arithmetic sum of the positive elements of C.

Case (iii): If B is the array $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$, SUM (B, DIM=1) is (/ 3, 7, 11 /) and SUM (B, DIM=2) is (/ 9, 12 /).

13.13.104 SYSTEM_CLOCK (COUNT, COUNT_RATE, COUNT_MAX).

Optional Arguments. COUNT, COUNT_RATE, COUNT_MAX

Description. Returns integer data from a real-time clock.

Class. Subroutine.

Arguments.

COUNT (optional) must be scalar and of type default integer. It is set to a processor-dependent value based on the current value of the basic clock or to –HUGE (0) if there is no clock. The processor-dependent value is incremented by one for each clock count until the value COUNT_MAX is reached and is reset to zero at the next count. It lies in the range 0 to COUNT_MAX if there is a clock.

COUNT_RATE (optional)
must be scalar and of type default integer. It is set to the number of basic clock counts per second, or to zero if there is no clock.

COUNT_MAX (optional)
must be scalar and of type default integer. It is set to the maximum value that COUNT can have, or to zero if there is no clock.

Example. If the basic system clock is a 24-hour clock that registers time in 1-second intervals, at 11:30 A.M. the reference

```
CALL SYSTEM_CLOCK (COUNT = C, COUNT_RATE = R, COUNT_MAX = M)
```

sets C = 11 × 3600 + 30 × 60 = 41400, R = 1, and M = 24 × 3600 − 1 = 86399.

13.13.105 TAN (X).

Description. Tangent function.

Class. Elemental function.

Argument. X must be of type real.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to tan(X), with X regarded as a value in radians.

Example. TAN (1.0) has the value 1.5574077 (approximately).

13.13.106 TANH (X).

Description. Hyperbolic tangent function.

Class. Elemental function.

Argument. X must be of type real.

Result Type and Type Parameter. Same as X.

Result Value. The result has a value equal to a processor-dependent approximation to tanh(X).

Example. TANH (1.0) has the value 0.76159416 (approximately).

13.13.107 TINY (X).

Description. Returns the smallest positive number in the model representing numbers of the same type and type parameter as the argument.

Class. Inquiry function.

Argument. X must be of type real. It may be scalar or array valued.

Result Type, Type Parameter, and Shape. Scalar with the same type and type parameter as X.

Result Value. The result has the value $b^{e_{min}-1}$ where b and e_{min} are as defined in 13.7.1 for the model representing numbers of the same type and type parameters as X.

Example. TINY (X) has the value 2^{-127} for real X whose model is as at the end of 13.7.1.

13.13.108 TRANSFER (SOURCE, MOLD, SIZE).

Optional Argument. SIZE

Description. Returns a result with a physical representation identical to that of SOURCE but interpreted with the type and type parameters of MOLD.

Class. Transformational function.

Arguments.

SOURCE	may be of any type and may be scalar or array valued.
MOLD	may be of any type and may be scalar or array valued.
SIZE (optional)	must be scalar and of type integer.

Result Type, Type Parameter, and Shape. The result is of the same type and type parameters as MOLD.

Case (i): If MOLD is a scalar and SIZE is absent, the result is a scalar.

Case (ii): If MOLD is array valued and SIZE is absent, the result is array valued and of rank one. Its size is as small as possible such that its physical representation is not shorter than that of SOURCE.

Case (iii): If SIZE is present, the result is array valued of rank one and size SIZE.

Result Value. If the physical representation of the result has the same length as that of SOURCE, the physical representation of the result is that of SOURCE. If the physical representation of the result is longer than that of SOURCE, the physical representation of the leading part is that of SOURCE and the remainder is undefined. If the physical representation of the result is shorter than that of SOURCE, the physical representation of the result is the leading part of SOURCE. If D and E are scalar variables such that the physical representation of D is as long as or longer than that of E, the value of TRANSFER (TRANSFER (E, D), E) must be the value of E. IF D is an array and E is an array of rank one, the value of TRANSFER (TRANSFER (E, D), E, SIZE (E)) must be the value of E.

Examples.

Case (i): TRANSFER (1082130432, 0.0) has the value 4.0 on a processor that represents the values 4.0 and 1082130432 as the string of binary digits 0100 0000 1000 0000 0000 0000 0000 0000.

Case (ii): TRANSFER ((/ 1.1, 2.2, 3.3 /), (/ (0.0, 0.0) /)) is a complex rank-one array of length two whose first element has the value (1.1, 2.2) and whose second element has a real part with the value 3.3. The imaginary part of the second element is undefined.

Case (iii): TRANSFER ((/ 1.1, 2.2, 3.3 /), (/ (0.0, 0.0) /), 1) has the value (/ (1.1, 2.2) /).

13.13.109 TRANSPOSE (MATRIX).

Description. Transpose an array of rank two.

Class. Transformational function.

Argument. MATRIX may be of any type and must have rank two.

Result Type, Type Parameters, and Shape. The result is an array of the same type and type parameters as MATRIX and with rank two and shape (*n*, *m*) where (*m*, *n*) is the shape of MATRIX.

Result Value. Element (*i*, *j*) of the result has the value MATRIX (*j*, *i*), $i = 1, 2, ..., n$; $j = 1, 2, ..., m$.

Example. If A is the array $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$, then TRANSPOSE (A) has the value $\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$.

13.13.110 TRIM (STRING).

Description. Returns the argument with trailing blank characters removed.

Class. Transformational function.

Argument. STRING must be of type character and must be a scalar.

Result Type and Type Parameters. Character with the same kind type parameter value of STRING and with a length that is the length of STRING less the number of trailing blanks in STRING.

Result Value. The value of the result is the same as STRING except any trailing blanks are removed. If STRING contains no nonblank characters, the result has zero length.

Example. TRIM (' A B ') has the value ' A B'.

13.13.111 UBOUND (ARRAY, DIM).

Optional Argument. DIM

Description. Returns all the upper bounds of an array or a specified upper bound.

Class. Inquiry function.

Arguments.

ARRAY	may be of any type. It 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.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \leq DIM \leq n$, where n is the rank of ARRAY.

Result Type, Type Parameter, and Shape. The result is of type default integer. It is scalar if DIM is present; otherwise, the result is an array of rank one and size n, where n is the rank of ARRAY.

Result Value.

Case (i): UBOUND (ARRAY, DIM) has a value equal to the upper bound for subscript DIM of ARRAY if dimension DIM of ARRAY does not have size zero and has the value zero if dimension DIM has size zero. For an array section or an array expression, its value is the number of elements in the corresponding dimension.

Case (ii): UBOUND (ARRAY) has a value whose ith component is equal to UBOUND (ARRAY, i), for $i = 1, 2, ..., n$, where n is the rank of ARRAY.

Example. If A is declared by the statement

```
REAL A (2:3, 7:10)
```

then UBOUND (A) is (/ 3, 10 /) and UBOUND (A, DIM=2) is 10.

13.13.112 UNPACK (VECTOR, MASK, FIELD).

Description. Unpack an array of rank one into an array under the control of a mask.

Class. Transformational function.

Arguments.

VECTOR	may be of any type. It must have rank one. Its size must be at least t where t is the number of true elements in MASK.
MASK	must be array valued and of type logical.
FIELD	must be of the same type and type parameters as VECTOR and must be conformable with MASK.

Result Type, Type Parameter, and Shape. The result is an array of the same type and type parameters as VECTOR and the same shape as MASK.

Result Value. The element of the result that corresponds to the ith true element of MASK, in array element order, has the value VECTOR (i) for $i = 1, 2, ..., t$, where t is the number of true values in MASK. Each other element has a value equal to FIELD if FIELD is scalar or to the corresponding element of FIELD if it is an array.

Example. Specific values may be "scattered" to specific positions in an array by using UNPACK. If M is the array $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$, V is the array (/ 1, 2, 3 /), and Q is the logical mask $\begin{bmatrix} . & T & . \\ T & . & . \\ . & . & T \end{bmatrix}$, where "T" represents true and "." represents false, then the result of UNPACK (V,

MASK=Q, FIELD=M) has the value $\begin{bmatrix} 0 & 2 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 3 \end{bmatrix}$ and the result of UNPACK (V, MASK=Q, FIELD=0) has the value $\begin{bmatrix} 0 & 2 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 3 \end{bmatrix}$.

UPPER_CASE useful

13.13.113 VERIFY (STRING, SET, BACK).

Optional Argument. BACK

Description. Verify that a set of characters contains all the characters in a string.

Class. Elemental function.

Arguments.

STRING	must be of type character.
SET	must be of type character.
BACK (optional)	must be of type logical.

START (optional)

Result Type and Type Parameter. Default integer.

Result Value. The value of the result is zero if each character in STRING is in SET or if STRING has zero length; otherwise, there are two cases as follows:

Case (i): If BACK is absent or present with the value .FALSE., the value of the result is the position of the leftmost character of STRING that is not in SET.

Case (ii): If BACK is present with the value .TRUE., the value of the result is the position of the rightmost character of STRING that is not in SET.

Examples. VERIFY ('ABBA', 'AB') has the value 0.

Case (i): VERIFY ('ABBA', 'A') has the value 2.

Case (ii): VERIFY ('ABBA', 'A', BACK = .TRUE.) has the value 3.

14. SCOPE, ASSOCIATION, AND DEFINITION

Each lexical token has a **scope**, which is either an executable program, a scoping unit, a single statement, or part of a statement. Within its scope, a lexical token has a single interpretation. An entity identified by a lexical token whose scope is an executable program is called a **global entity**. An entity identified by a lexical token whose scope is a scoping unit (2.2.1) is called a **local entity**. An entity identified by a lexical token whose scope is a single statement or part of a statement is called a **statement entity**.

By means of association, a named entity may be known by the same name or a different name in a different scoping unit, or by a different name in the same scoping unit.

14.1 Scope of Names. The names of external procedures, common blocks, and program units have a scope of an executable program.

The names of variables, constants, statement functions, internal procedures, module procedures, dummy procedures, intrinsic procedures, keyword arguments, derived types, type components, namelist groups, and constructs have a scope of a scoping unit.

The name of a variable that appears as a dummy argument in a statement function statement has a scope of the statement in which it appears.

The name of a variable that appears as the DO variable of an implied-DO in a DATA statement or an array constructor has a scope of the implied-DO list.

14.1.1 Global Entities. Program units, common blocks, and external procedures are global entities of an executable program. A name that identifies a global entity must not be used to identify any other global entity in the same executable program.

14.1.2 Local Entities. Within a scoping unit, entities in the following classes:

(1) Named variables, named constants, named constructs, statement functions, internal procedures, module procedures, dummy procedures, intrinsic procedures, derived types, and namelist group names,

(2) Type components, in a separate class for each type, and

(3) Argument keywords, in a separate class for each procedure with an explicit interface

are local entities of that scoping unit.

A name that identifies a global entity in a scoping unit must not be used to identify a local entity of class (1) in that scoping unit, except for a common block name (14.1.2.1) or ~~an external~~ function name (14.1.2.2).

Within a scoping unit, a name that identifies a local entity of one class must not be used to identify another entity of the same class, except in the case of overloaded procedures (14.1.2.3). A name that identifies a local entity of one class may be used to identify a local entity of another class.

The name of a local entity identifies that entity in a scoping unit and may be used to identify any local or global entity in another scoping unit.

14.1.2.1 Common Blocks. A common block name in a scoping unit also may be the name of any local entity other than a named constant, intrinsic function, or a local variable that is also ~~an external~~ function in a function subprogram. If a name is used for both a common block and a local entity, the appearance of that name in any context other than as a common block name in a COMMON or SAVE statement identifies only the local entity. Note that an intrinsic function name may be a common block name in a scoping unit that does not reference the intrinsic function.

14.1.2.2 Function Results. If a function subprogram does not have a RESULT clause in its function statement, there must be a local variable with the same name as the function. If a function subprogram contains an ENTRY statement, there must be a local variable with the same name as the entry.

14.1.2.3 Procedure Overloading. Within a scoping unit, two procedures may be referenced by the same name provided they both have explicit interfaces and at least one of them has a nonoptional dummy argument which

(1) Corresponds by position in the argument list to a dummy argument not present in the other, present with a different type, present with different type parameters where neither is an asterisk, or present with a different rank when both are pointers or assumed-shape arrays; and

(2) Corresponds by argument keyword to a dummy argument not present in the other, present with a different type, present with different type parameters where neither is an asterisk, or present with a different rank when both are pointers or assumed-shape arrays.

For example, the procedures with interfaces

```
INTERFACE A
   SUBROUTINE AR (X)
      REAL X
   END SUBROUTINE AR

   SUBROUTINE AI (I)
      INTEGER I
   END SUBROUTINE AI
END INTERFACE
```

satisfy rules (1) and (2) and therefore the procedures may be overloaded. However, if I were declared REAL, rule (1) would not be satisfied while rule (2) remains satisfied; this case is not allowed because the reference to A in the statement

```
CALL A (0.0)
```

would be ambiguous.

14.1.2.4 Components. A component name has the same scope as the type of which it is a component. It may appear only within a designator of a component of a structure of that type. If the type is accessible in another scoping unit by use association or host association (14.6.1.2) and the definition of the type does not contain the PRIVATE statement (4.4.1), the component name is accessible for names of components of structures of that type in that scoping unit.

14.1.2.5 Argument Keywords. A dummy argument name in an internal procedure, module procedure, or a procedure interface block has a scope as an argument keyword of the scoping unit of its host. As an argument keyword, it may appear only in a procedure reference for the procedure of which it is a dummy argument. If the procedure or procedure interface block is accessible in another scoping unit by use association or host association (14.6.1.2), the argument keyword is accessible for procedure references for that procedure in that scoping unit.

14.1.3 Statement Entities. The name of a variable that appears as a dummy argument in a statement function statement has a scope of the statement in which it appears. It has the type and type parameters that it would have if it were the name of a variable in the scoping unit that includes the statement function.

The name of a variable that appears as the DO variable of an implied-DO in a DATA statement or an array constructor has a scope of the implied-DO list. It has the type that it would have if it were the name of a variable in the scoping unit that includes the DATA statement or array constructor and this type must be integer.

The name of a statement entity also may be the name of a global or local entity in the same scoping unit; in this case, the name is interpreted within its statement scope as that of the statement variable.

14.2 Scope of Labels. A label has a scope of a scoping unit. No two statements in the same scoping unit may have the same label.

Scope of a construct name?

14.3 Scope of External Input/Output Units. An external input/output unit has the scope of an executable program.

14.4 Scope of Operators. The intrinsic operators have a scope of an executable program. A defined operator has a scope of a scoping unit. Within a scoping unit, two operations may be identified by the same operator provided they have at least one corresponding operand with different type, different type parameters where neither is an asterisk, or different rank.

14.5 Scope of the Assignment Symbol. Intrinsic assignment has a scope of an executable program. A defined assignment has a scope of a scoping unit. Within a scoping unit, two assignments may be identified by the assignment symbol provided they have at least one corresponding operand with different type, different type parameters where neither is an asterisk, or different rank.

14.6 Association. Two entities may become associated by name association, pointer association, or storage association.

14.6.1 Name Association. There are three forms of **name association**: argument association, use association, and host association. Argument, use, and host association provide mechanisms by which entities known in a scoping unit may be accessed in another scoping unit.

14.6.1.1 Argument Association. The rules governing argument association are given in Section 12. As explained in 12.4, execution of a procedure reference establishes an association between an actual argument and its corresponding dummy argument. Argument association may be sequence association (12.4.1.4).

The name of the dummy argument may be different from the name, if any, of its associated actual argument. (Note that an actual argument may be a nameless data entity, such as an expression that is not simply a variable or constant.) The dummy argument name is the name by which the associated actual argument is known, and by which it may be accessed, in the referenced procedure.

Upon termination of execution of a procedure reference, all argument associations established by that reference are terminated. A dummy argument of that procedure may be associated with an entirely different actual argument in a subsequent execution of the procedure.

14.6.1.2 Use Association and Host Association. Use association is the association of names in different scoping units specified by a USE statement. The rules for use association are given in 11.3.2. They allow for the renaming of the entities being accessed.

The rules for host association are given in 11.2.2.

Use association or host association allows access in one scoping unit to entities defined in another scoping unit and remains in effect throughout the execution of the executable program.

Within a scoping unit, an entity accessed by use association or host association must not appear in a type declaration statement or otherwise have any of its attributes specified. It assumes all attributes, and only those attributes, of its associated entity. If the entity is renamed in a USE statement in a scoping unit, the original name is not associated with it in this scoping unit and may be used for other purposes. The new local name may be used in exactly the same way as the original name could have been used if there had been no renaming and no conflict with another local name.

Table 14.2 Summary Comparison of Use and Host Associations.

Characteristic	Use Associations	Host Associations
Scope	Single scoping unit, plus using scoping units if in a module	Single scoping unit plus scoping units of internal or module subprograms
Duration	Entire program execution	Entire program execution
May change?	No	No
How established?	Appearance in USE statement	Internal or module subprogram or derived-type definition
How terminated?	Termination of execution of the executable program	Termination of execution of the executable program
Appearance in USE statement	Normal (only) way to establish	Not allowed
Allowed with unallocated parent	Yes	Yes
May appear in ALLOCATE statement	Yes	Yes
May appear in NULLIFY or DEALLOCATE statement	Yes	Yes

14.6.2 Pointer Association. Pointer association between a pointer and a target allows the target to be referenced or defined by way of the pointer. A pointer may be associated with different targets or disassociated at different times during execution of a program.

The initial association status of a pointer is undefined. The association status of a pointer becomes defined as associated when the pointer is allocated (6.3.1) or pointer assigned to a target (7.5.2). The association status of a pointer becomes defined as disassociated when the pointer is nullified (6.3.2), pointer assigned to a disassociated pointer, or deallocated (6.3.3). Only the target of a previously allocated pointer may be deallocated.

The pointer association status of a pointer becomes undefined if the associated target ceases to exist (12.4.1.1).

A pointer that is associated with a definable target may be defined or undefined according to the same rules as for a variable (14.7).

14.6.3 Storage Association. Storage sequences are used to describe relationships that exist among variables, array elements, substrings, common blocks, and arguments. **Storage association** is the association of two or more data objects that occurs when two or more storage sequences share or are aligned with one or more storage units.

14.6.3.1 Storage Sequence. A **storage sequence** is a sequence of zero, one, or more storage units. The **size of a storage sequence** is the number of storage units in the storage sequence. A **storage unit** is a character storage unit or a numeric storage unit.

A scalar object of type default integer, default real, or default logical has a storage sequence of one numeric storage unit.

A structure has no storage sequence. SEQUENCE attribute ??

A structure component, or an element of it if an array, has no storage sequence, even if of an intrinsic type.

An integer, real, logical, character, or complex object with explicitly specified kind type parameter has no storage sequence.

A scalar object of type double precision real or default complex has a storage sequence of two numeric storage units. In a complex storage sequence, the real part has the first storage unit and the imaginary part has the second storage unit.

A scalar object of type default character has a storage sequence of zero, one, or more character storage units. The number of character storage units in the storage sequence is the length of the character entity. The order of the sequence corresponds to the ordering of character positions (4.3.2.1 and 5.1.1.5).

A sequence of storage sequnces is itself a storage sequence. The order of the storage units in such a composite storage sequence is that of the individual storage units in each of the constituent storage sequences taken in succession, ignoring any zero-sized constituent sequences.

Each common block has a storage sequence (5.5.2.1).

Each data object appearing in a storage association context has a storage sequence (2.4.9).

14.6.3.2 Association of Storage Sequences. Two nonzero-sized storage sequences s_1 and s_2 are **storage associated** if the ith storage unit of s_1 is the same as the jth storage unit of s_2. This causes the $(i+k)$th storage unit of s_1 to be the same as the $(j+k)$th storage unit of s_2, for each integer k such that $1 \le i+k \le$ *size of* s_1 and $1 \le j+k \le$ *size of* s_2.

Storage association also is defined between two zero-sized storage sequences, and between a zero-sized storage sequence and a storage unit. If a zero-sized storage sequence occurs in a sequence of storage sequences and is not the last sequence, it is storage associated with its successor. If the successor is another zero-sized storage sequence, the two sequences are associated. If the successor is a nonzero-sized storage sequence, the zero-sized sequence is storage associated with the first storage unit of the successor. Two storage units that are each associated with the same zero-sized storage sequence are the same storage unit.

what if it is last

14.6.3.3 Association of Scalar Data Objects. Two scalar data objects are storage associated if their storage sequences are storage associated. Two scalar entities are **totally associated** if they have the same storage sequence. Two scalar entities are **partially associated** if they are associated but not totally associated.

The definition status and value of a data object affects the definition status and value of any associated entity. An EQUIVALENCE statement, a COMMON statement, an ENTRY statement, or a procedure reference may cause association of storage sequences.

how?

An EQUIVALENCE statement causes association of data objects only within one scoping unit, unless one of the equivalenced entities is also in a common block (5.5.1.1 and 5.5.2.1).

COMMON statements cause data objects in one scoping unit to become associated with data objects in another scoping unit.

In a function subprogram, an ENTRY statement causes the entry name to become associated with the name of the function subprogram which appears in the FUNCTION statement.

or the result name?

Partial association may exist only between two default character entities or between a default complex or double precision real entity and an entity of type default integer, default real, default logical, double precision real, or default complex.

Except for character entities, partial association may occur only through the use of COMMON, EQUIVALENCE, or ENTRY statements. Partial association must not occur through argument association, except for arguments of type default character.

In the example:

```
REAL A (4), B
COMPLEX C (2)
DOUBLE PRECISION D
EQUIVALENCE (C (2), A (2), B), (A, D)
```

the third storage unit of C, the second storage unit of A, the storage unit of B, and the second storage unit of D are specified as the same. The storage sequences may be illustrated as:

```
Storage unit       1     2     3     4     5
                 ----C(1)----|---C(2)----
                       A(1)  A(2)  A(3)  A(4)
                             --B--
                           ------D------
```

A(2) and B are totally associated. The following are partially associated: A(1) and C(1), A(2) and C(2), A(3) and C(2), B and C(2), A(1) and D, A(2) and D, B and D, C(1) and D, and C(2) and D. Note that although C(1) and C(2) are each associated with D, C(1) and C(2) are not associated with each other.

Partial association of character entities occurs when some, but not all, of the storage units of the entities are the same. In the example:

```
CHARACTER A*4, B*4, C*3
EQUIVALENCE (A (2:3), B, C)
```

A, B, and C are partially associated.

14.7 Definition and Undefinition of Variables. A variable may be defined or may be undefined and its definition status may change during execution of an executable program. An action that causes a variable to become undefined does not imply that the variable was previously defined. An action that causes a variable to become defined does not imply that the variable was previously undefined.

14.7.1 Definition of Objects and Subobjects. Arrays, including sections, and variables of derived, character, complex, or double precision real type are objects that consist of zero or more subobjects. Associations may be established between variables and subobjects and between subobjects of different variables. These subobjects may become defined or undefined.

(1) An object is defined if and only if all of its subobjects are defined.

(2) If an object is undefined, at least one (but not necessarily all) of its subobjects are undefined.

14.7.2 Variables That Are Always Defined. Zero-sized arrays and zero-length strings are always defined.

14.7.3 Variables That Are Initially Defined. The following variables are defined initially:

(1) Variables specified to have initial values by DATA statements,

(2) Variables specified to have initial values by type declaration statements, and

(3) Variables that are always defined.

14.7.4 Variables That Are Initially Undefined. All other variables are initially undefined.

14.7.5 Events That Cause Variables to Become Defined. Variables become defined as follows:

(1) Execution of an assignment statement other than a masked array assignment statement causes the variable that precedes the equals to become defined.

(2) Execution of a masked array assignment statement may cause some or all of the array elements in the assignment statement to become defined (7.5.3).

(3) As execution of an input statement proceeds, each variable that is assigned a value from the input file becomes defined at the time that data is transferred to it. (See (5) in 14.7.6.)

(4) Execution of a DO statement causes the DO variable, if any, to become defined.

(5) Beginning of execution of the action specified by an implied-DO list in an input/output statement causes the implied-DO variable to become defined.

(6) Execution of an ASSIGN statement causes the variable in the statement to become defined with a statement label value.

(7) A reference to a procedure causes a subobject of a dummy argument to become defined if the corresponding subobject of the actual argument is defined with a value that is not a statement label.

(8) Execution of an input/output statement containing an input/output IOSTAT= specifier causes the specified integer variable to become defined.

(9) Execution of an input/output statement containing a NULLS= specifier causes the specified integer variable to become defined.

(10) Execution of an input/output statement containing a SIZE= specifier causes the specified integer variable to be defined.

(11) Execution of an INQUIRE statement causes any variable that is assigned a value during the execution of the statement to become defined if no error condition exists.

(12) When a character storage unit becomes defined, all associated character storage units become defined.

When a numeric storage unit becomes defined, all associated numeric storage units of the same type become defined, except that variables associated with the variable in an ASSIGN statement become undefined when the ASSIGN statement is executed.

When a scalar variable without a storage sequence becomes defined, all associated variables become defined.

what if it has a storage sequence

(13) When a default complex entity becomes defined, all partially associated default real entities become defined.

(14) When both parts of a default complex entity become defined as a result of partially associated default real or default complex entities becoming defined, the default complex entity becomes defined.

(15) Execution of an ALLOCATE or DEALLOCATE statement with a STAT= specifier causes the variable specified by the STAT= specifier to become defined.

(16) Allocation of a zero-sized array causes the array to become defined.

(17) Invocation of a procedure causes any automatic object of zero size in that procedure to become defined.

(18) Execution of a pointer assignment statment that associates a pointer with a target that is defined.

14.7.6 Events That Cause Variables to Become Undefined. Variables become undefined as follows:

(1) When a variable of a given type becomes defined, all associated variables of different type become undefined. However, when a variable of type default real is partially associated with a variable of type default complex, the complex variable does not become undefined when the real variable becomes defined and the real variable does not become undefined when the complex variable becomes defined. When a variable of type default complex is partially associated with another variable of type default

complex, definition of one does not cause the other to become undefined.

(2) Execution of an ASSIGN statement causes the variable in the statement to become undefined as an integer. Variables that are associated with the variable also become undefined.

(3) If the evaluation of a function may cause an argument of the function or a variable in a module or in a common block to become defined and if a reference to the function appears in an expression in which the value of the function is not needed to determine the value of the expression, the argument or variable becomes undefined when the expression is evaluated.

(4) The execution of a RETURN statement or an END statement within a subprogram causes all variables local to its scoping unit or local to the current instance of its scoping unit for a recursive invocation to become undefined except for the following:

(a) Variables with the SAVE attribute.

(b) Variables in blank common.

(c) Variables in a named common block that appears in the subprogram and appears in at least one other scoping unit that is making either a direct or indirect reference to the subprogram.

(d) Variables accessed from the host scoping unit.

(e) Variables accessed from a module that also is accessed in a scoping unit that is currently in execution.

(f) Initially defined entities that neither have been redefined nor have become undefined.

(5) When an error condition or end-of-file condition occurs during execution of an input statement, all of the variables specified by the input list or *namelist-group* of the statement become undefined.

(6) When an error or end-of-file condition occurs during execution of an input/output statement, some or all of the implied-DO variables may become undefined (9.4.3).

(7) Execution of a defined assignment statement may leave all or part of the variable that precedes the equals undefined.

(8) Execution of a direct access input statement that specifies a record that has not been written previously causes all of the variables specified by the input list of the statement to become undefined.

(9) Execution of an INQUIRE statement may cause the NAME=, RECL=, and NEXTREC= variables to become undefined (9.6).

(10) When a character storage unit becomes undefined, all associated character storage units become undefined.

When a numeric storage unit becomes undefined, all associated numeric storage units become undefined unless the undefinition is a result of defining an associated numeric storage unit of different type (see (1) above).

When a scalar variable without a storage sequence becomes undefined, all associated variables become undefined.

(11) A reference to a procedure causes part of a dummy argument to become undefined if the corresponding part of the actual argument is defined with a value that is a statement label value.

(12) When an allocatable object is nullified or deallocated, it becomes undefined. Successful execution of an ALLOCATE statement causes the allocated object to become undefined.

(13) Execution of an INQUIRE statement causes all inquiry specifier variables to become undefined if an error condition exists, except for the variable in the IOSTAT= specifier,

if any.

(14) When a procedure is invoked:

(a) An optional dummy argument that is not associated with an actual argument is undefined.

(b) A dummy argument with INTENT (OUT) is undefined.

(c) An actual argument associated with a dummy argument with INTENT (OUT) becomes undefined.

(d) A subobject of a dummy argument is undefined if the corresponding subobject of the actual argument is undefined.

APPENDIX A. FORTRAN FAMILY OF STANDARDS

(This appendix is not part of American National Standard X3.9-198x, but is included for information only.)

A host language standard, such as Fortran, should take responsibility for coordinating other standards built on its base to prevent the development of conflicting collateral standards. A Fortran Reference Model has been suggested for the **Fortran Family of Standards**.

The Fortran Family of Standards consists of:

(1) The Fortran Language Standard

(2) Supplementary Standards based on Procedure Libraries

(3) Supplementary Standards based on Module Libraries

(4) Secondary Standards

X3.9-1978 (the previous Fortran standard) is referred to as **FORTRAN 77** in this appendix. This standard is referred to as **Fortran 8x**. A possible successor is referred to as **Fortran 9x**.

A.1 The Fortran Language Standard. The Fortran Language consists of **primary features** from FORTRAN 77, **decremental features** that are deleted or obsolescent in this standard, and **incremental features** that add new constructs to Fortran. (See Figure A.1.)

A.1.1 Primary Features. These features are those from the FORTRAN 77 standard that continue to be useful and characteristic of the language. Primary features are expected to continue throughout the life of Fortran or at least for the next several revisions of the language.

A.1.2 Incremental Features. These features are new to the language and are needed to improve the usefulness of Fortran. They are developed from current practice in extended Fortran implementations and in other contemporary languages.

The criteria for incremental features are:

(1) The feature is responsive to new system architectures.

(2) The feature improves the functionality of Fortran.

(3) The feature is desirable for certain important special purpose applications.

(4) The feature's inclusion enhances portability.

(5) The feature uses modern language technology.

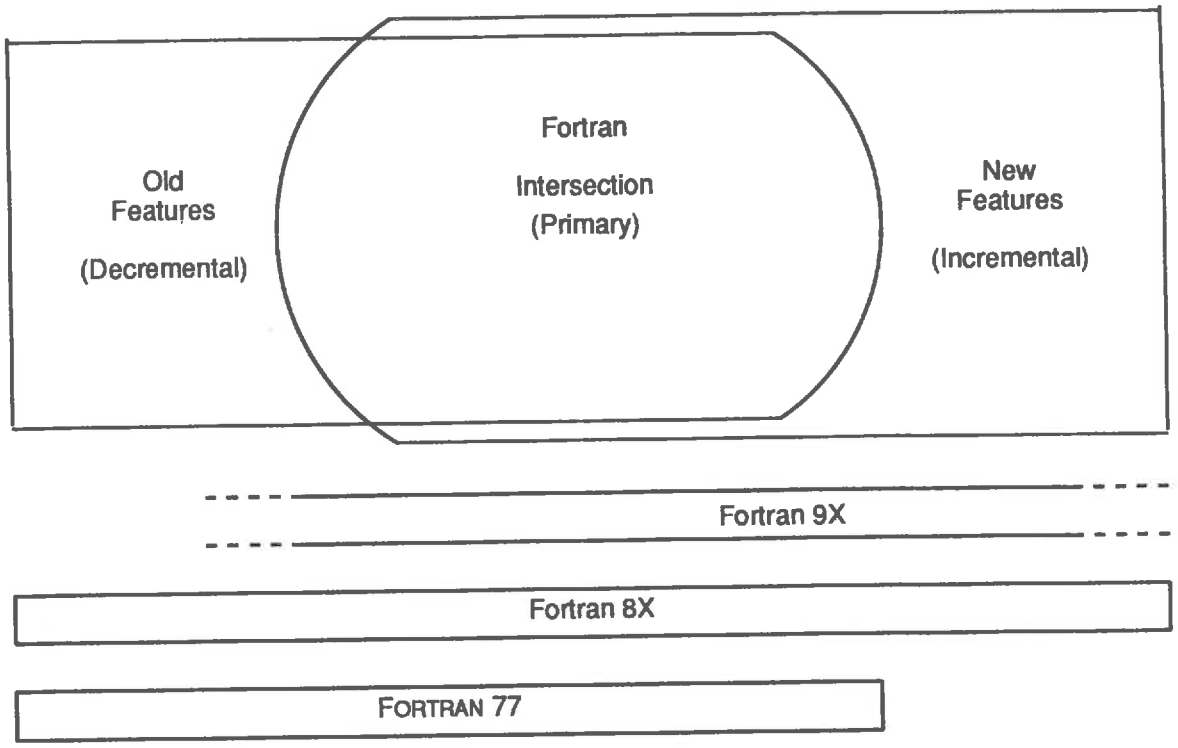
(6) The feature is compatible with the primary and decremental features.

A.1.3 Decremental Features. Decremental features are those features that are deleted or obsolescent in the Fortran Standard. Obsolescent features are candidates for removal from future versions of the Fortran Standard. Marking a feature as obsolescent does not imply its removal from subsequent standards; notification is given that these features *may* be removed in subsequent revisions.

Appendix B further describes decremental features.

A.1.4 Compatibility. All of FORTRAN 77 is included within Fortran 8x. Fortran 8x consists of the complete language of primary, incremental, and decremental features. No segmentation or subsetting of the language is implied. FORTRAN 77 is the combination of the primary features and the decremental features. Programs written in FORTRAN 77 are compatible with Fortran 8x and, with few exceptions, incremental features may be added to existing FORTRAN 77 programs.

FORTRAN FAMILY OF STANDARDS
(Reference Model)



Core Fortran is

Primary plus Incremental Features

Figure A.1 The Fortran Language Standard.

A.1.5 Core. Core Fortran is the combination of the primary features and incremental features.

A.2 Supplementary Standards Based on Procedure Libraries. Supplementary Standards add functionality to the Fortran language by using the interface mechanisms specified in the Fortran Language Standard. Examples of supplementary standards are the Industrial Real Time Fortran (IRTF) specification and the Fortran binding to the Graphical Kernel System (GKS). These are standards themselves and conform with the FORTRAN 77 standard. Other possible candidates for supplementary standards might be the standardization of certain utility or mathematical libraries and the standardization of data base facilities. While a supplementary standard adds functionality to the Fortran Family, it does not alter the syntax of constructs in Fortran.

A.2.1 Interface Mechanisms. A supplementary standard based on procedure references is called a **procedure supplementary standard**. Such standards must use the interface mechanisms provided in Fortran to describe specific definitions of a process. The interface mechanisms provided in FORTRAN 77 are limited to procedure references. Fortran 8x extends this interface capability by allowing keywords and optional arguments in procedure references.

A.3 Supplementary Standards Based on Module Libraries. A supplementary standard based on modules is called a **module supplementary standard**. Supplementary standards may specify modules that provide a high level of application-oriented functionality. These may include the definition of new data types and their accompanying operators. Modules are nonexecutable program units containing definitions made available to any other program unit by the USE statement. Many problem-oriented applications would make excellent candidates for module

supplementary standards. Modules may be included in the Fortran Standard document or they may be standardized in separate documents.

A.3.1 Interface Mechanisms. The interface mechanisms provided in Fortran 8x contain a set of facilities for binding a variety of additional features, such as graphics, to Fortran. These facilities include modules which make definitions, data declarations, and procedure libraries available to an executable program. The USE statement provides the means for referencing specific modules. Supplementary standards may use these mechanisms in defining a specific process within the Fortran Family of Standards.

SUPPLEMENTARY STANDARDS

Fortran Family of Standards

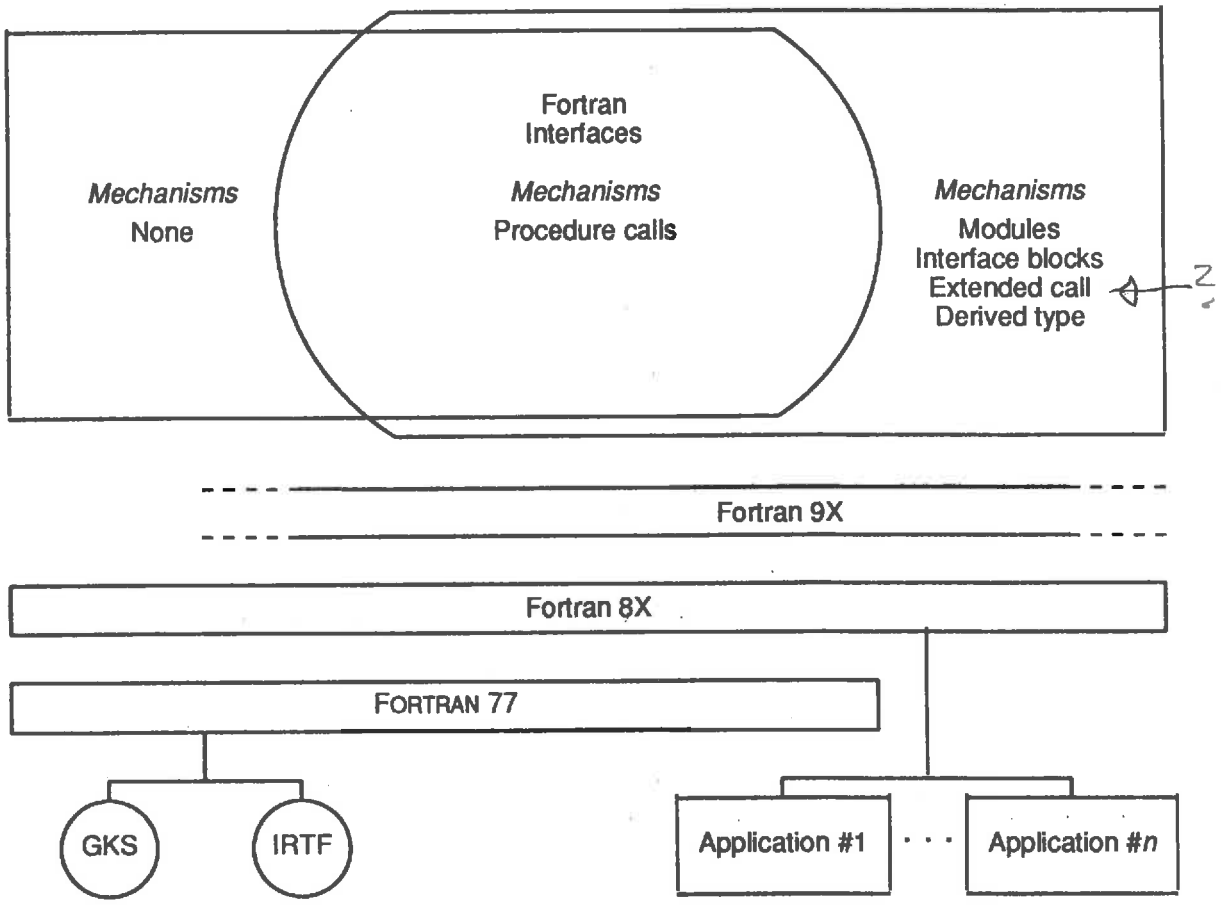


Figure A.2 Supplementary Standards.

SECONDARY STANDARDS

Fortran Family of Standards

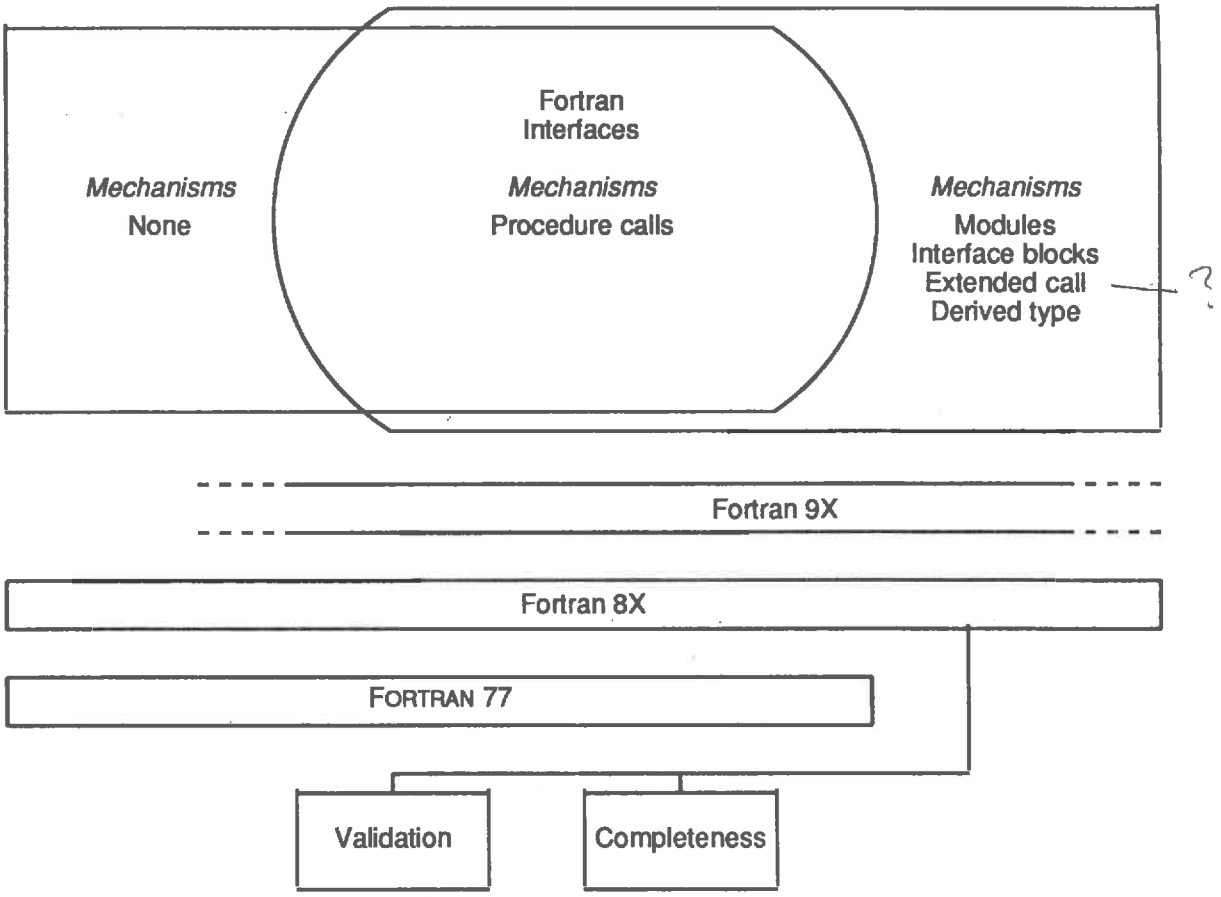


Figure A.3 Secondary Standards.

A.4 Rules for Supplementary Standards. Some rules governing the preparation of supplementary standards based on procedure and module libraries are:

(1) A module may be appended to the Fortran Standard or it may be a separate standard.

(2) If a module is appended to the Fortran Standard, it is forwarded for review at the same time as the standard. If it is a separate supplementary standard, there is an independent standardization process.

(3) A module is not part of the Standard. It is a member of the Fortran Family of Standards.

(4) Standard modules must not use obsolescent features (i.e., must conform to the Fortran Core.) When the Fortran Standard is revised, a formerly standard-conforming module may cease to be standard conforming because of the use of (old) decremental features.

(5) When the Fortran Standard is revised, a review may determine that modifications are needed to take advantage of any new functionality (incremental features) in the standard.

(6) A name registration for supplementary standards is available from the Fortran Standards Technical Subcommittee.

(7) Separate standards projects should be defined for each supplementary and secondary standard. Task groups may be formed within the Fortran Standards Technical Subcommittee for development of supplementary and secondary standards.

(8) Standard Modules prepared outside the committee and its task groups must use the interface mechanisms in the language. Requests for new facilities in the Fortran Standard must be processed by the Fortran Standards Technical Subcommittee.

(9) The Fortran Standards Technical Subcommittee should review all candidates for supplementary and secondary standards to determine if they are standard conforming. This must be done in a timely manner.

A.5 Secondary Standards. Secondary standards do not impact or change the syntax of the language nor do they change the semantics of the Fortran Standard. Instead, these standards may make requirements on the conformance of programs using the Fortran Standard. For example, certain constructs that control the execution sequence of a program may be required to flag specific conditions that occur during execution. Validation of programs during compilation or execution is another example. Conformance requirements could be expanded in a separate secondary standard. The syntax rules used to help describe the form that Fortran statements take are included in the Fortran Standard (1.5). These rules are described in a variation of BNF. Currently, there are no secondary standards in the Fortran Family of Standards; however, work is proceeding in these areas for Fortran and for programming languages in general. See Figure A.3.

A.6 Standard Conformance. Any program unit containing syntax not defined in the Fortran language is not standard conforming with respect to the Fortran Standard. The inclusion of a USE statement does not make the nonstandard syntax become standard conforming. A program unit that uses only syntax and semantics defined in the Fortran language standard and one or more standard modules is standard conforming with respect to the Fortran Family of Standards.

In moving to a revised standard, a number of features rather than the complete standard are often selected by implementors. It is recommended that partial implementations of major features not be done. For example, if the array facilities are to be included, as many of the array features as possible should be implemented.

A.6.1 Name Registration. A list of names registered with the Fortran Standards Technical Subcommittee will be kept for reference by those who are preparing a module intended for the Fortran Family of Standards.

A.7 Fortran Family of Standards. Figure A.4 is the complete diagram of the Fortran Family of Standards. It includes the Fortran language with incremental, decremental, and primary features. The interface mechanisms shown refer to the procedure and module supplementary standards in the reference model.

FORTRAN FAMILY OF STANDARDS

(Reference Model)

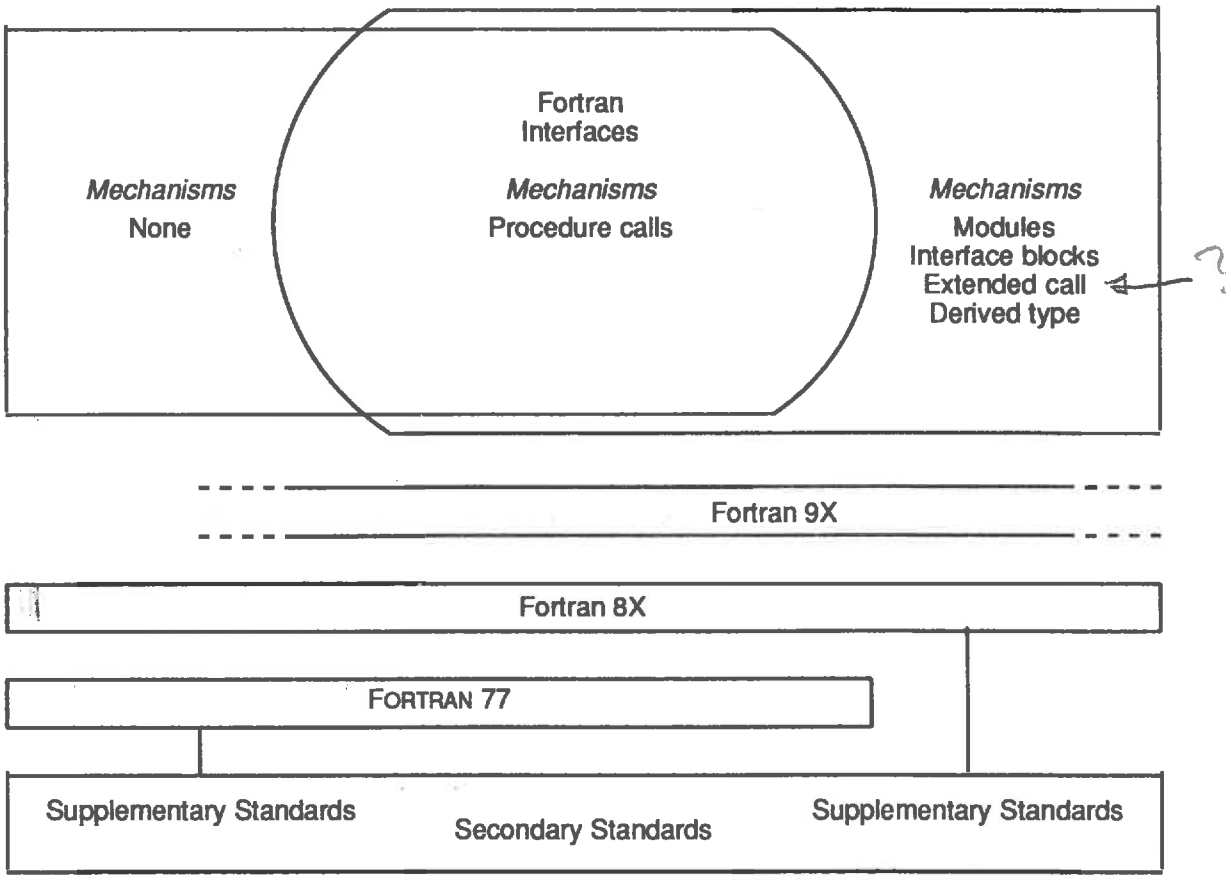


Figure A.4 The Fortran Family of Standards.

APPENDIX B. DECREMENTAL FEATURES

(This appendix is not part of American National Standard X3.9-198x, but is included for information only.)

This appendix more fully describes the rationale for the specific decremental (deleted or obsolescent) features (1.6). Possible alternatives to the obsolescent features are described.

B.1 Deleted Features. The deleted features are those features of FORTRAN 77 that are redundant and considered largely unused. Section 1.6.1 describes the nature of the deleted features. The list of deleted features in this standard is empty.

B.2 Obsolescent Features. The obsolescent features are those features of FORTRAN 77 that are redundant and for which better methods are available in FORTRAN 77. Section 1.6.2 describes the nature of obsolescent features. The obsolescent features are:

(1) Arithmetic IF — use the IF statement (8.1.2.4) or IF construct (8.1.2)

(2) Real and double precision DO control variables and DO loop control expressions — use integer (8.1.4.1)

(3) Shared DO termination and termination on a statement other than END DO or CONTINUE — use an END DO or a CONTINUE statement for each DO statement

(4) Branching to an END IF statement from outside its IF block — branch to the statement following the END IF

(5) Alternate return — see B.2.1

(6) PAUSE statement — see B.2.2

(7) ASSIGN and assigned GO TO statements — see B.2.3

(8) Assigned FORMAT specifiers — see B.2.4

B.2.1 Alternate Return. An alternate return introduces labels into an argument list to allow the called procedure to direct the execution of the caller upon return. The same effect can be achieved with a return code that is used in a computed GO TO statement on return. This avoids an irregularity in the syntax and semantics of argument association. For example,

```
CALL SUBR_NAME (X, Y, Z, *100, *200, *300)
```

may be replaced by

```
CALL SUBR_NAME (X, Y, Z, RETURN_CODE)
GO TO (100, 200, 300) RETURN_CODE
```

B.2.2 PAUSE Statement. Execution of a PAUSE statement requires operator or system-specific intervention to resume execution. In most cases, the same functionality can be achieved as effectively and in a more portable way with the use of an appropriate READ statement that awaits some input data.

B.2.3 ASSIGN and Assigned GO TO Statements. The ASSIGN statement allows a label to be dynamically assigned to an integer variable, and the assigned GO TO statement allows "indirect branching" through this variable. This hinders the readability of the program flow, especially if the integer variable also is used in arithmetic operations. The two totally different usages of the integer variable can be an obscure source of error.

The statements may be replaced by ordinary assignment and the computed GO TO. For example,

```
ASSIGN 100 TO I
...
GO TO I (100, 200, 300)
```

may be replaced by

```
I = 1
...
GO TO (100, 200, 300) I
```

B.2.4 Assigned FORMAT Specifiers. The ASSIGN statement also allows the label of a FORMAT statement to be dynamically assigned to an integer variable, which can later be used as a format specifier in READ, WRITE, or PRINT statements. This hinders readability, permits inconsistent usage of the integer variable, and can be an obscure source of error.

This functionality was provided in FORTRAN 77 via character variables, arrays, and constants.

or internal subprograms

APPENDIX C. SECTION NOTES

(This appendix is not part of American National Standard X3.9-198x, but is included for information only.)

C.1 Section 1 Notes.

C.1.1 Conformance (1.4). The standard requires a standard-conforming processor to be capable of detecting and reporting the use within a program unit of forms designated as deleted or obsolescent and of additional forms or relationships, where such use can be detected by reference to the numbered syntax rules and their associated constraints. It is recommended that the processor be accompanied by documentation that specifies the limits it imposes on the size and complexity of a program and the means of reporting when these limits are exceeded, that defines the additional forms and relationships it allows, and that defines the means of reporting the use of additional forms and relationships and the use of deleted or obsolescent forms. Note that in this context, the use of a deleted form is the use of an additional form.

C.1.2 Obsolescence (1.6.2). Use of obsolescent features is discouraged. ~~Each~~ Any obsolescent feature may be considered for deletion in the next revision of the Fortran standard.

C.2 Section 2 Notes. Argument keywords can make procedure references more readable and allow actual arguments to be in any order. This latter property permits optional arguments. (2.5.2)

C.3 Section 3 Notes.

C.3.1 Collating Sequence (3.1.7). A partial collating sequence is specified. If possible, a processor should use the American National Standard Code for Information Interchange, ANSI X3.4-1977 (ASCII), sequence for the complete Fortran character set.

C.3.2 Comment Lines (3.3.1.1, 3.3.2.1). The standard does not restrict the number of consecutive comment lines. The limit of 19 continuation lines or 2640 characters permitted for a statement should not be construed as being a limitation on the number of consecutive comment lines.

C.3.3 Statement Labels (3.2.5). There are 99999 unique statement labels and a processor must accept 99999 as a statement label. However, a processor may have an implementation limit on the total number of unique statement labels in one program unit.

C.3.4 Source Form (3.3). In fixed source form, an exclamation point (!) in character position 6 is interpreted as a continuation indicator unless it appears within commentary indicated by a "C" or "*" in character position 1 or by another "!" in character positions 1–5. (3.3.2.3)

The source form of FORTRAN 77, FORTRAN 66, and the initial Fortran in 1954 was predicated on a common form of input, the 80-column card. However, on the IBM 704, only 72 columns could be used and the remaining eight columns were designated as commentary. In some implementations of FORTRAN 77, these columns are so used. They contain "line numbers" and are used by an editor to manage changes to a program. (3.3.2)

In developing Fortran 8x, the Fortran Standards Technical Subcommittee X3J3 sought to eliminate the FORTRAN 77 restriction on source line size. X3J3 believes that 66 positions are inadequate to represent readable Fortran source code, particularly with "long" names and the use of indentation.

Given the need for an incompatible new source form in Fortran 8x, X3J3 relaxed other restrictions of the rigid card form. Positions six and seven are no longer "special" and the continuation mark is on the line being continued rather than on the continuation line. Blank characters are generally significant in the new source form, but other features of the new form apply to either form, and are allowed in either. (3.3.1)

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

C.4 Section 4 Notes.

C.4.1 Zero (4.3.1). A processor must not consider a negative zero to be different from a positive zero.

C.4.2 Intrinsic and Derived Data Types (4.3, 4.4). ANSI X3.9-1978 provided only data types explicitly defined in the standard (logical, integer, real, double precision, complex, and character). This standard provides those intrinsic types and provides derived types to allow the creation of new data types. A derived-type definition specifies a data structure consisting of components of intrinsic types and of other derived types. Such a type definition does not represent a data object, but rather, a template for declaring named objects of that derived type. For example, the definition

```
TYPE POINT
   INTEGER X_COORD
   INTEGER Y_COORD
END TYPE POINT
```

specifies a new derived type named POINT which is composed of two components of intrinsic type integer (X_COORD and Y_COORD). The statement TYPE (POINT) FIRST, LAST declares two data objects, FIRST and LAST, that can hold values of type POINT.

X3.9-1978 provided REAL and DOUBLE PRECISION intrinsic types as approximations to mathematical real numbers. This standard generalizes REAL as an intrinsic type with a type parameter that selects the approximation method. The type parameter is named KIND and has values that are processor dependent. DOUBLE PRECISION is treated as a synonym for REAL (*k*), where *k* is the implementation-defined kind type parameter value KIND (0.0D0).

Real literal constants may be specified with a kind type parameter to ensure that they have a particular kind type parameter value (4.3.1.2).

For example, with the specifications

```
INTEGER Q
PARAMETER (Q = 8)
REAL (Q) B
```

the literal constant 10.93_Q has the same precision as the variable B.

X3.9-1978 did not allow zero length character strings. They are permitted by this standard (4.3.2.1).

Objects are of different derived type if they are declared using different derived-type definitions. For example,

```
TYPE APPLES
   INTEGER NUMBER
END TYPE APPLES
TYPE ORANGES
   INTEGER NUMBER
END TYPE ORANGES
TYPE (APPLES) COUNT1
TYPE (ORANGES) COUNT2
COUNT 1 = COUNT2 ! ERRONEOUS STATEMENT MIXING APPLES AND ORANGES
```

Even though all components of objects of type APPLES and objects of type ORANGES have identical intrinsic types, the objects are of different types because they were declared using different derived-type definitions.

C.4.3 Selection of the Approximation Methods. This standard permits the selection of the real approximation method for an entire program to be parameterized through the use of the parameterized real data type and module. This is accomplished by defining a named integer constant, say FLOAT, to have a specific kind type parameter value, and to use that named constant in all real, complex, and derived-type declarations. For example, the specification statements

```
INTEGER FLOAT
PARAMETER (FLOAT = 8)
REAL (FLOAT) X, Y
COMPLEX (FLOAT) Z
```

specifies that the approximation method corresponding to a kind type parameter value of 8 is supplied for the data objects X, Y, and Z in the program unit. The kind type parameter value FLOAT can be made available to an entire program by placing the INTEGER and PARAMETER specification statements in a module and accessing the named constant FLOAT with a USE statement. Note that by changing 8 to 4 once in the module, a different approximation method is selected.

To avoid the use of the processor-dependent values 4 or 8, replace 8 by KIND (0.0) or KIND (0.0D0). Another way to avoid these processor-dependent values is to select the kind value using the intrinsic inquiry function SELECTED_REAL_KIND. This function, given integer arguments P and R specifying minimum requirements for decimal precision and decimal exponent range, respectively, returns the kind type parameter value of the approximation method that has at least P decimal digits of precision and at least a range for positive numbers of 10^{-R} and 10^{R}. In the above specification statement, the 8 may be replaced by, for instance, SELECTED_REAL_KIND (10, 50), which requires an approximation method to be selected with at least 10 decimal digits of precision and an exponent range from 10^{-50} to 10^{50}.

C.4.4 Components and Storage of Derived Types (4.4.1). A structure resolves into a sequence of components of intrinsic type. The use of this terminology in no way implies that these components are stored in this, or any other, order. Nor is there any requirement that contiguous storage be used. The sequence merely refers to the fact that in writing the definitions there will necessarily be an order in which the components appear, and this will define a sequence of components. This order is of limited significance since a component of an object of derived type will always be accessed by a component name except in the following contexts: the sequence of expressions in a derived-type value constructor, the data values in namelist input data, and the inclusion of the structure in an input/output list of a formatted data transfer, where it is expanded to this sequence of components. Provided the processor adheres to the defined order in these cases, it is otherwise free to organize the storage of the components for any structure in memory as best suited to the particular architecture. What about SEQUENCE

C.4.5 Pointers. This standard introduces pointers as names that can change dynamically their association with a target object. In a sense, a normal variable is a name with a fixed association with a specific object. A normal variable name refers to the same storage space throughout the lifetime of a variable. A pointer name may refer to different storage space, or even no storage space, at different times. A variable may be considered to be a descriptor for space to hold values of the appropriate type, type parameters, and array rank such that the values stored in the descriptor are fixed when the variable is created by its declaration. A pointer also may be considered to be a descriptor, but one whose values may be changed dynamically so as to describe different pieces of storage. When a pointer is declared, space to hold the descriptor is created, but the space for the target object is not.

A derived type may have one or more components that are defined to be pointers. It may have a component that is a pointer to an object of the same derived type. This "recursive" data definition allows dynamic data structures such as linked lists, trees, and graphs to be constructed. For example,

```
TYPE CELL           ! Define a "recursive" type
   INTEGER :: VAL
   TYPE (CELL), POINTER :: NEXT_CELL
```

```
END TYPE CELL

TYPE (CELL), TARGET :: HEAD
TYPE (CELL), POINTER :: CURRENT, TEMP  ! Declare pointers
INTEGER :: IOEM, K

HEAD % VAL = 0
CURRENT => HEAD                        ! CURRENT points to head of list
DO
   READ (*, *, IOSTAT = IOEM) K        ! Read next value, if any
   IF (IOEM <> 0) EXIT
   ALLOCATE (TEMP)                     ! Create new cell each iteration
   TEMP % VAL = K                      ! Asssign value to cell
   CURRENT % NEXT_CELL => TEMP         ! Attach new cell to list
   CURRENT => TEMP                     ! CURRENT points to new end of list
END DO
```

A list is now constructed and the last linked cell contains a disassociated pointer. A loop can be used to "walk through" the list.

```
CURRENT => HEAD
DO
   WRITE (*, *) CURRENT % VAL
   IF (.NOT. ASSOCIATED (CURRENT % NEXT_CELL)) EXIT
   CURRENT => CURRENT % NEXT_CELL
END DO
```

C.4.6 The POINTER Attribute (5.1.2.7). The pointer attribute must be specified to declare a pointer. The type, type parameters, and rank that must be specified at the same time determine the characteristics of the target objects that can be associated with the pointers declared in the statement. An obvious model for interpreting declarations of pointers is that such declarations create for each name a descriptor. Such a descriptor includes all the data necessary to describe fully and locate in memory an object and all subobjects of the type, type parameters, and rank specified. The descriptor is created empty; it does not contain values describing how to access an actual memory space. These descriptor values will be filled in when the pointer is associated with actual target space.

The following example illustrates the use of pointers in an iterative algorithm:

```
PROGRAM DYNAM_ITER
   REAL, DIMENSION (:, :), POINTER :: A, B, SWAP  ! Declare pointers
   ...
   READ (*, *) N, M
   ALLOCATE (A (N, M), B (N, M))  ! Allocate target arrays
   ! Read values into A
   ...
   ITER: DO
      ...
      ! Apply  transformation of values in A to produce values in B
      ...
      IF (CONVERGED) EXIT ITER
      ! Swap A and B
      SWAP => A; A => B; B => SWAP
   END DO ITER
   ...
END
```

C.5 Section 5 Notes.

C.5.1 Type Declaration Statements (5.1). Type declaration statements in X3.9-1978 required different attributes of an entity to be specified in different statements (INTEGER, SAVE, DATA, etc.). This standard allows the attributes of an entity to be specified in a single extended form of the type statement. For example,

```
INTEGER, DIMENSION (10, 10), SAVE :: A, B, C
REAL, PARAMETER :: P1 = 3.14159265, E = 2.718281828
```

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

If kind is omitted from a REAL declaration, the objects are of default real type. This corresponds to the FORTRAN 77 real type (5.1.1.2).

C.5.2 The TARGET Attribute. The TARGET attribute must be specified for any nonpointer object that may, during the execution of the program, become associated with a pointer. This attribute is defined entirely for optimization purposes. It allows the processor to assume that any nonpointer object not explicitly declared as a target may be referred to only by way of its original declared name. In particular, it means that implicitly-declared objects must not be used as pointer targets. This will allow a processor to perform optimizations that otherwise would not be possible in the presence of certain pointers.

The following example illustrates the use of the TARGET attribute in an iterative algorithm:

```
PROGRAM ITER
   REAL, DIMENSION (1000, 1000), TARGET :: A, B
   REAL, DIMENSION (:,:), POINTER       :: IN, OUT, SWAP
   ...
   ! Read values into A
   ...
   IN => A              ! Associate IN with target A
   OUT => B             ! Associate OUT with target B
   ...
   ITER:DO
      ...
      ! Apply transformation of IN values to produce OUT
      ...
      IF (CONVERGED) EXIT ITER
      ! Swap IN and OUT
      SWAP => IN; IN => OUT; OUT => SWAP
   END DO ITER
   ...
END
```

C.6 Section 6 Notes.

C.6.1 Substrings (6.1.1). Substrings are of zero length when the starting point exceeds the ending point. This was not allowed in FORTRAN 77. This standard also allows substrings of literal character constants and named character constants.

C.6.2 Array Element References (6.2.2). A subscript reference to an element outside the declared bounds is not standard conforming, as in FORTRAN 77.

C.6.3 Structure Components (6.1.2). Components of a structure are referenced by writing the components of successive levels of the structure hierarchy until the desired component is described. For example,

```
TYPE ID_NUMBERS
   INTEGER SSN
   INTEGER EMPLOYEE_NUMBER
END TYPE ID_NUMBERS

TYPE PERSON_ID
   CHARACTER (LEN=30) LAST_NAME
   CHARACTER (LEN=1) MIDDLE_INITIAL
   CHARACTER (LEN=30) FIRST_NAME
   TYPE (ID_NUMBERS) NUMBER
END TYPE PERSON_ID

TYPE PERSON
   INTEGER AGE
   TYPE (PERSON_ID) ID
END TYPE PERSON

TYPE (PERSON) GEORGE, MARY

PRINT *, GEORGE % AGE              ! PRINT THE AGE COMPONENT
PRINT *, MARY % ID % LAST_NAME     ! PRINT LAST_NAME OF MARY
PRINT *, MARY % ID % NUMBER % SSN ! PRINT SSN OF MARY
PRINT *, GEORGE % ID % NUMBER      ! PRINT SSN AND EMPLOYEE_NUMBER OF GEORGE
```

A structure component may be a data object of intrinsic type as in the case of GEORGE%AGE or it may be of derived type as in the case of GEORGE%ID%NUMBER. The resultant component may be a scalar or an array of intrinsic or derived type.

```
TYPE LARGE
   INTEGER ELT (10)
   INTEGER VAL
END TYPE LARGE

TYPE (LARGE) A (5)         ! 5 ELEMENT ARRAY EACH OF WHOSE ELEMENTS INCLUDES
                           ! A 10 ELEMENT ARRAY ELT AND A SCALAR VAL.
PRINT *, A (1)             ! PRINTS 10 ELEMENT ARRAY ELT AND SCALAR VAL.
PRINT *, A (1) % ELT (3) ! PRINTS SCALAR ELEMENT 3 OF ARRAY ELEMENT 1 OF A.
PRINT *, A (2:4) % VAL   ! PRINTS SCALAR VAL FOR ARRAY ELEMENTS 2 TO 4 OF A.
```

C.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 that is the pointer. An ALLOCATE is assumed to create space for a suitable object and to "assign" to the pointer the values necessary to describe that space. A NULLIFY breaks the association of the pointer with the space. A DEALLOCATE breaks the association and releases the space. Depending on the implementation, it could be seen as setting a flag in the pointer that indicates whether the values in the descriptor are valid, or it could clear the descriptor values to some (say zero) value indicative of the pointer not currently pointing to anything. A pointer assignment copies the values necessary to describe the space occupied by the target into the descriptor that is the pointer. Descriptors are copied, values of objects are not.

If PA and PB are both pointers and PB currently is associated with a target, then

```
PA => PB
```

results in PA being associated with the same target as PB. If PB was disassociated, then PA becomes disassociated.

The standard is defined so that such associations are direct and independent. A subsequent statement

```
PB => D
```

or

```
ALLOCATE(PB)
```

has no effect on the association of PA with its target. A statement

```
DEALLOCATE(PB)
```

leaves PA as a "dangling pointer" to space that has been released. The program must not use PA again until it becomes associated via pointer assignment or an ALLOCATE statement.

DEALLOCATE should only be used to release space that was created by a previous ALLOCATE. Thus the following is invalid:

```
REAL, TARGET :: T
REAL, POINTER :: P
...
P => T
DEALLOCATE(P) ! Not allowed: P's target was not allocated
```

The basic principle is that ALLOCATE, DEALLOCATE, NULLIFY, and pointer assignment primarily affect the pointer rather than the target. ALLOCATE creates a new target but, other than breaking its connection with the specified pointer, it has no effect on the old target. Neither NULLIFY nor pointer assignment has any effect on targets. A given piece of memory that was allocated and associated with a pointer will become inaccessible to a program if the pointer is nullified and no other pointer was associated with this piece of memory. Such pieces of memory may be reused by the processor if this is expedient. However, whether such inaccessible memory is in fact reused is entirely processor dependent.

C.7 Section 7 Notes.

C.7.1 Character Assignment. The FORTRAN 77 restriction that none of the character positions being defined in the character assignment statement may be referenced in the expression has been removed (7.5.1.5).

C.7.2 Evaluation of Function References. If more than one function reference appears in a statement, they may be executed in any order (subject to a function result being evaluated after the evaluation of its arguments) and their values must not depend on the order of execution. This lack of dependence on order of evaluation permits parallel execution of the function references (7.1.7.1).

C.7.3 Pointers in Expressions. A pointer is basically considered to be like any other variable when it is used as a primary in an expression. If a pointer is used as an operand to an operator that expects a value the pointer will automatically deliver the value contained in the space currently described by the pointer, that is, the value of the target object currently associated with the pointer. In value-demanding expression contexts, pointers are dereferenced.

C.7.4 Pointers on the Left Side of an Assignment. A pointer that appears on the left of an intrinsic assignment statement also is dereferenced and is taken to be referring to the space that is its current target. Therefore, the assignment statement specifies the normal copying of the value of the right-hand expression into this target space. All the normal rules of intrinsic assignment hold; the type and type parameters of the expression and the pointer target must agree and the shapes must be conformable.

For intrinsic assignment of derived types, nonpointer components are assigned and pointer components are pointer assigned. Dereferencing is applied only to entire scalar objects, not

selectively to pointer subobjects.

For example, suppose a type such as

```
TYPE CELL
   INTEGER :: VAL
   TYPE(CELL), POINTER :: NEXT_CELL
ENDTYPE
```

is defined and objects such as HEAD and CURRENT are declared using

```
TYPE(CELL), TARGET :: HEAD
TYPE(CELL), POINTER :: CURRENT
```

If a linked list has been created and attached to HEAD and the pointer CURRENT has been allocated space, statements such as

```
CURRENT = HEAD
CURRENT = CURRENT % NEXT_CELL
```

cause the contents of the cells referenced on the right to be copied to the cell referred to by CURRENT. In particular, the right-hand side of the second statement causes the pointer component in the cell, CURRENT, to be selected. This pointer is dereferenced since it is in an expression context to produce the target's integer value and a pointer to a cell that is contained in the target's NEXT_CELL component. The left-hand side causes the pointer CURRENT to be dereferenced to produce its present target, namely space to hold a cell (an integer and a cell pointer). The integer value on the right is copied to the integer space on the left and the pointer components are pointer assigned (the descriptor on the right is copied into the space for a descriptor on the left). When a statement such as

```
CURRENT => CURRENT % NEXT_CELL
```

is executed, the descriptor value in CURRENT % NEXT_CELL is copied to the descriptor named CURRENT. In this case, CURRENT is made to point at a different target.

In the intrinsic assignment statement, the space associated with the current pointer does not change but the values stored in that space do. In the pointer assignment, the current pointer is made to associate with different space. Using the intrinsic assignment causes a linked list of cells to be moved up through the current "window"; the pointer assignment causes the current pointer to be moved down through the list of cells.

C.8 Section 8 Notes.

C.8.1 Loop Control. Fortran provides several forms of loop control:

(1) With an iteration count and a DO variable. This is the classic Fortran DO.

(2) DO while a logical expression is true.

(3) DO "forever".

C.8.2 The CASE Construct. At most one case block is selected within a CASE construct, and there is no fall-through from one block into another block within a CASE construct. Thus there is no requirement for the user to exit explicitly from a block.

C.8.3 Examples of Invalid DO Constructs. The following are all examples of invalid skeleton DO constructs:

Example 1:

```
DO I = 1, 10
   ...
END DO LOOP    ! No matching construct name
```

Example 2:

```
LOOP: DO 1000 I = 1, 10    ! No matching construct name
   ...
1000  CONTINUE
```

Example 3:

```
LOOP1: DO
   ...
END DO LOOP2    ! Construct names don't match
```

Example 4:

```
DO I = 1, 10    ! Label required or ...
   ...
1010 CONTINUE   ! ... END DO required
```

Example 5:

```
DO 1020 I = 1, 10
   ...
1021 END DO         ! Labels don't match
```

Example 6:

```
FIRST: DO I = 1, 10
   SECOND: DO J = 1, 5
      ...
   END DO FIRST    ! Improperly nested DOs
END DO SECOND
```

C.9 Section 9 Notes.

C.9.1 Input/Output Records (9.1). What is called a "record" in Fortran is commonly called a "logical record". There is no concept in Fortran of a "physical record".

An endfile record does not necessarily have any physical embodiment. The processor may use a record count or other means to register the position of the file at the time an ENDFILE statement is executed, so that it can take appropriate action when that position is reached again during a read operation. The endfile record, however it is implemented, is considered to exist for the BACKSPACE statement (9.1.3).

C.9.2 Files (9.2). This standard accommodates, but does not require, file cataloging. To do this, several concepts are introduced.

C.9.2.1 File Connection (9.3). Before any input/output can be performed on a file, it must be connected to a unit. The unit then serves as a designator for that file as long as it is connected. To be connected does not imply that "buffers" have or have not been allocated, that "file-control tables" have or have not been filled out, or that any other method of implementation has been used. Connection means that (barring some other fault) a READ or WRITE statement can be executed on the unit, hence on the file. Without a connection, a READ or WRITE statement cannot be executed.

C.9.2.2 File Existence (9.2.1.1). Totally independent of the connection state is the property of existence, this being a file property. The processor "knows" of a set of files that exist at a given time for a given executable program. This set would include tapes ready to read, files in a catalog, a keyboard, a printer, etc. The set may exclude files inaccessible to the executable program because of security, because they are already in use by another executable program, etc. This standard does not specify which files exist, hence wide latitude is available to a processor to implement security, locks, privilege techniques, etc. Existence is a convenient concept to designate all of the files that an executable program can potentially process.

All four combinations of connection and existence may occur:

Connect	Exist	Examples
Yes	Yes	A card reader loaded and ready to be read
Yes	No	A printer before the first line is written
No	Yes	A file named 'JOAN' in the catalog
No	No	A reel of tape destroyed in the fire last week

Means are provided to create, delete, connect, and disconnect files.

C.9.2.3 File Names. A file may have a name. The form of a file name is not specified. If a system does not have some form of cataloging or tape labeling for at least some of its files, all file names will disappear at the termination of execution. This is a valid implementation. Nowhere does this standard require names to survive for any period of time longer than the execution time span of an executable program. Therefore, this standard does not impose cataloging as a prerequisite. The naming feature is intended to allow use of a cataloging system where one exists.

C.9.2.4 File Access (9.2.1.2). This standard does not address problems of security, protection, locking, and many other concepts that may be part of the concept of "right of access". Such concepts are considered to be in the province of an operating system.

The OPEN and INQUIRE statements can be extended naturally to consider these things.

Possible access methods for a file are: sequential and direct. The processor may implement two different types of files, each with its own access method. It might also implement one type of file with two different access methods.

Direct access to files is of a simple and commonly available type, that is, fixed-length records. The key is a positive integer.

C.9.2.5 Nonadvancing Input/Output (9.2.1.3.1). An obvious model for nonadvancing output is using unbuffered input/output. This standard has been constructed to allow the processor to output the portion of the record immediately. This standard does not explicitly discuss the generation of an end-of-record during nonadvancing output but it should implicitly occur before the generation of an endfile record due to a BACKSPACE, ENDFILE, REWIND, or CLOSE statement.

For nonadvancing input, the processor is not required to read partial records. The processor may read the entire record into an internal buffer and make successive portions of the record available to successive input statements.

C.9.3 OPEN Statement (9.3.4). A file may become connected to a unit in either of two ways: preconnection or execution of an OPEN statement. Preconnection is performed prior to the beginning of execution of an executable program by means external to Fortran. For example, it may be done by job control action or by processor established defaults. Execution of an OPEN statement is not required to access preconnected files (9.3.3).

The OPEN statement provides a means to access existing files that are not preconnected. An OPEN statement may be used in either of two ways: with a file name (open by name) and without a file name (open by unit). A unit is given in either case. Open by name connects the specified file to the specified unit. Open by unit connects a processor-determined default file to the specified unit. (The default file may or may not have a name.)

Therefore, there are three ways a file may become connected and hence processed: preconnection, open by name, and open by unit. Once a file is connected, there is no means in standard

Fortran to determine how it became connected.

An OPEN statement may also be used to create a new file. In fact, any of the foregoing three connection methods may be performed on a file that does not exist. When a unit is preconnected, writing the first record creates the file. With the other two methods, execution of the OPEN statement creates the file.

When an OPEN statement is executed, the unit specified in the OPEN may or may not already be connected to a file. If it is already connected to a file (either through preconnection or by a prior OPEN), then omitting the FILE= specifier in the OPEN statement implies that the file is to remain connected to the unit. Such an OPEN statement may be used to change the values of the BLANK=, DELIM=, or PAD= specifiers.

Note that, since an OPEN that specifies STATUS = 'SCRATCH' is not allowed to have a FILE= specifier, such an OPEN always attempts to retain any connection that the specified unit may have. If the unit were already connected to a file, and if that connection did not have a STATUS of SCRATCH, then the OPEN would be illegal because the value of the STATUS= specifier must not be changed by the OPEN.

The following examples illustrate these rules. In the first example, unit 10 is preconnected to a SCRATCH file; the OPEN statement changes the value of PAD= to YES.

```
CHARACTER (LEN = 20) CH1
WRITE (10, '(A)') 'THIS IS RECORD 1'
OPEN (UNIT = 10, STATUS = 'SCRATCH', PAD = 'YES')
REWIND 10
READ (10, '(A20)') CH1    ! CH1 now has the value
                          ! 'THIS IS RECORD 1    '
```

In the next example, we first connect unit 12 to a file named FRED, with a status of OLD. The second OPEN statement then opens unit 12 again, retaining the connection to the file FRED, but changing the value of the DELIM= specifier to QUOTE.

```
CHARACTER (LEN = 25) CH2, CH3
OPEN (12, FILE = 'FRED', STATUS = 'OLD', DELIM = 'NONE')
CH2 = '"THIS STRING HAS QUOTES."'
              ! Quotes in string CH2
WRITE (10, *) CH2           ! Written with no delimiters
OPEN (12, DELIM = 'QUOTE')  ! Now quote is the delimiter
REWIND 12
READ (12, *) CH3  ! CH3 now has the value
                  ! 'THIS STRING HAS QUOTES.  '
```

The next example is invalid because it attempts to change the value of the STATUS= specifier.

```
OPEN (10, FILE = 'FRED', STATUS = 'OLD')
WRITE (10, *) A, B, C
OPEN (10, STATUS = 'SCRATCH')    ! Attempts to make FRED
                                 ! a SCRATCH file
```

The previous example could be made valid by closing the unit first, as in the next example.

```
OPEN (10, FILE = 'FRED', STATUS = 'OLD')
WRITE (10, *) A, B, C
CLOSE (10)
OPEN (10, STATUS = 'SCRATCH')    ! Opens a different
                                 ! SCRATCH file
```

C.9.4 Connection Properties (9.3.2). When a unit becomes connected to a file, either by execution of an OPEN statement or by preconnection, the following connection properties may be established:

(1) An access method, which is sequential or direct, is established for the connection (9.3.4.3).

(2) A form, which is formatted or unformatted, is established for a connection to a file that exists or is created by the connection. For a connection that results from execution of an OPEN statement, a default form (which depends on the access method, as described in 9.2.1.2) is established if no form is specified. For a preconnected file that exists, a form is established by preconnection. For a preconnected file that does not exist, a form may be established, or the establishment of a form may be delayed until the file is created (for example, by execution of a formatted or unformatted WRITE statement) (9.3.4.4).

(3) A record length may be established. If the access method is direct, the connection established a record length, which specifies the length of each record of the file. An existing file with records that are not all of equal length must not be connected for direct access.

If the access method is sequential, records of varying lengths are permitted. In this case, the record length established specifies the maximum length of a record in the file (9.3.4.5).

(4) A blank significance property, which is ZERO or NULL, is established for a connection for which the form is formatted. This property has no effect on output. For a connection that results from execution of an OPEN statement, the blank significance property is NULL by default if no blank significance property is specified. For a preconnected file, the property is NULL.

The blank significance property of the connection is effective at the beginning of each formatted input statement. During execution of the statement, any BN or BZ edit descriptors encountered may temporarily change the effect of embedded and trailing blanks (9.3.4.6).

FORTRAN 77 did not define default values for the blank significance properties of internal and preconnected files. This standard defines the default values for these files to be NULL, matching that of files connected by the OPEN statement.

A processor has wide latitude in adapting these concepts and actions to its own cataloging and job control conventions. Some processors may require job control action to specify the set of files that exist or that will be created by an executable program. Some processors may require no job control action prior to execution. This standard enables processors to perform a dynamic open, close, and file creation, but it does not require such capabilities of the processor.

The meaning of "open" in contexts other than Fortran may include such things as mounting a tape, console messages, spooling, label checking, security checking, etc. These actions may occur upon job control action external to Fortran, upon execution of an OPEN statement, or upon execution of the first read or write of the file. The OPEN statement describes properties of the connection to the file and may or may not cause physical activities to take place. It is a place for an implementation to define properties of a file beyond those required in standard Fortran.

C.9.5 CLOSE Statement (9.3.5). Similarly, the actions of dismounting a tape, protection, etc. of a "close" may be implicit at the end of a run. The CLOSE statement may or may not cause such actions to occur. This is another place to extend file properties beyond those of standard Fortran. Note, however, that the execution of a CLOSE statement on a unit followed by an OPEN statement on the same unit to the same file or to a different file is a permissible sequence of events. The processor must not deny this sequence solely because the implementation chooses to do the physical act of closing the file at the termination of execution of the program.

C.9.6 INQUIRE Statement. Table C.1 indicates the values assigned to the INQUIRE statement specifier variables when no error condition is encountered during execution of the INQUIRE statement

Table C.1 Values Assigned to INQUIRE Specifier Variables.

Specifier	INQUIRE by File		INQUIRE by Unit	
	Unconnected	Connected	Connected	Unconnected
EXIST=	.TRUE. if file exists, .FALSE. otherwise		.TRUE. if unit exists, .FALSE. otherwise	
OPENED=	.FALSE.	.TRUE.		.FALSE.
NUMBER=	–1	unit no.		–1
NAMED=	.TRUE. if file named, .FALSE. otherwise			.FALSE.
NAME=	filename (may not be same as FILE= value)		filename if named, else undefined	undefined
ACCESS=	UNDEFINED	SEQUENTIAL or DIRECT		UNDEFINED
SEQUENTIAL=	YES, NO, or UNKNOWN			UNKNOWN
DIRECT=	YES, NO, or UNKNOWN			UNKNOWN
FORM=	UNDEFINED	FORMATTED or UNFORMATTED		UNDEFINED
FORMATTED=	YES, NO, or UNKNOWN			UNKNOWN
UNFORMATTED=	YES, NO, or UNKNOWN			UNKNOWN
RECL=	undefined	if direct access, record length; else maximum record length		undefined
NEXTREC=	undefined	if direct access, next record #; else undefined		undefined
BLANK=	UNDEFINED	NULL, ZERO, or UNDEFINED		UNDEFINED
DELIM=	UNDEFINED	APOSTROPHE, QUOTE, NONE, or UNDEFINED		UNDEFINED
PAD=	YES	YES or NO		YES
POSITION=	UNDEFINED	REWIND, APPEND, ASIS, or UNDEFINED		UNDEFINED
ACTION=	UNDEFINED	READ, WRITE, or READ/WRITE		UNDEFINED
IOLENGTH=	RECL= value for *output-item-list*			

C.9.7 Keyword Specifiers. Keyword forms of specifiers are used because there are many specifiers and a positional notation is difficult to remember. The keyword form sets a style for processor extensions. The UNIT= and FMT= keywords are offered for completeness, but their use is optional. Thus, compatibility with ANSI X3.9-1966 and ANSI X3.9-1978 is achieved.

C.9.8 Format Specifications (9.4.1.1). Format specifications may be included in the READ and WRITE statements, as in:

```
READ (UNIT = 10, FMT = '(I3, A4, F10.2)') K, ALPH, X
```

C.9.9 Unformatted Input/Output (9.4.4.4.1). Unformatted input/output involving derived-type list items forms the single exception to the rule that the appearance of an aggregate list item (such as an array) is equivalent to the appearance of its expanded list of component parts. This exception permits the processor greater latitude in improving efficiency or in matching the processor-dependent sequence of values for a derived-type object to similar sequences for aggregate objects used by means other than Fortran. However, formatted input/output of all list items and unformatted input/output of list items other than those of derived types adhere to the above rule.

C.9.10 Input/Output Restrictions. An example of a restriction on input/output statements (9.8) is that an input statement must not specify that data are to be read from a printer.

C.9.11 Pointers in an Input/Output List. Data transfers always involve the movement of values between a file and internal space. A pointer as such cannot be read or written. A pointer may, therefore, appear as an item in an input/output list if it is currently associated with a target that can receive a value (input) or can deliver a value (output). A derived type object with one or more pointer components must not appear as an item in an input/output list because the value of a pointer component is the descriptor for a location in memory. As such, this has no processor-independent representation external to the processor.

C.10 Section 10 Notes.

C.10.1 Character Constant Format Specification (10.1.2, 10.7.1). If a character constant is used as a format specifier in an input/output statement, care must be taken that the value of the character constant is a valid format specification. In particular, if a format specification delimited by apostrophes contains an apostrophe edit descriptor, two apostrophes must be written to delimit the apostrophe edit descriptor and four apostrophes must be written for each apostrophe that occurs within the apostrophe edit descriptor. For example, the text:

```
 2 ISN'T 3
```

may be written by various combinations of output statements and format specifications:

```
      WRITE (6, 100) 2, 3
100 FORMAT (1X, I1, 1X, 'ISN''T', 1X, I1)

      WRITE (6, '(1X, I1, 1X, ''ISN''''T'', 1X, I1)') 2, 3

      WRITE (6, '(A)') ' 2 ISN''T 3'
```

Note that doubling of internal apostrophes usually may be avoided by using quotation marks to delimit the format specification and doubling of internal quotation marks usually may be avoided by using apostrophes as delimiters.

C.10.2 T Edit Descriptor (10.6.1.1). The T edit descriptor includes the carriage control character in lines that are to be printed. T1 specifies the carriage control character and T2 specifies the first character that is printed.

C.10.3 Length of Formatted Records. The length of a formatted record is not always specified exactly and may be processor dependent.

C.10.4 Number of Records (10.3, 10.4, 10.6.2). The number of records read by an explicitly formatted input statement can be determined from the following rule: A record is read at the beginning of the format scan (even if the input list is empty), at each slash edit descriptor encountered in the format, and when a format rescan occurs at the end of the format.

The number of records written by an explicitly formatted output statement can be determined from the following rule: A record is written when a slash edit descriptor is encountered in the format, when a format rescan occurs at the end of the format, and at completion of execution of the

output statement (even if the output list is empty). Thus, the occurrence of *n* successive slashes between two other edit descriptors causes *n* – 1 blank lines if the records are printed. The occurrence of *n* slashes at the beginning or end of a complete format specification causes *n* blank lines if the records are printed. However, a complete format specification containing *n* slashes (*n* > 0) and no other edit descriptors causes *n* + 1 blank lines if the records are printed. For example, the statements

```
  PRINT 3
3 FORMAT (/)
```

will write two records that cause two blank lines if the records are printed.

C.10.5 List-Directed Input/Output (10.8). List-directed input/output allows data editing according to the type of the list item instead of by a format specifier. It also allows data to be free-field, that is, separated by commas or blanks.

If no list items are specified in a list-directed input/output statement, one input record is skipped or one empty output record is written.

C.10.6 List-Directed Input (10.8.1). The following examples illustrate list-directed input. A blank character is represented by b.

Example 1:

Program:

```
J = 3
READ *, I
READ *, J
```

Sequential input file:

record 1: `b1b,4bbbbb`
record 2: `,2bbbbbbbb`

Result: I = 1, J = 3.

Explanation: The second READ statement reads the second record. The initial comma in the record designates a null value; therefore, J is not redefined.

Example 2:

Program:

```
CHARACTER A *8, B *1
READ *, A, B
```

Sequential input file:

record 1: `'bbbbbbbb'`
record 2: `'QXY'b'Z'`

Result: A = 'bbbbbbbb', B = 'Q'

Explanation: In the first record, the rightmost apostrophe is interpreted as delimiting the constant (it cannot be the first of a pair of embedded apostrophes representing a single apostrophe because this would involve the prohibited "splitting" of the pair by the end of a record); therefore, A is set to the character constant 'bbbbbbbb'. The end of a record acts as a blank, which in this case is a value separator because it occurs between two constants.

C.11 Section 11 Notes.

C.11.1 Main Program and Block Data Program Unit (11.1, 11.4). The name of the main program or of a block data program unit has no explicit use within the Fortran language. It is available for documentation and for possible use within a computer environment.

A processor may implement an unnamed main program or unnamed block data program unit assigning it a default name. However, this name must not conflict with any other global name in a standard-conforming executable program. This might be done by making the default name one which is not permitted in a standard-conforming program (for example, by including a character not normally allowed in names) or by providing some external mechanism such that for any given program the default name can be changed to one that is otherwise unused.

C.11.2 Dependent Compilation (11.3). This standard, like its predecessors, is intended to permit the implementation of conforming processors in which a program can be broken into multiple units, each of which can be separately translated in preparation for execution. Such processors are commonly described as supporting separate compilation. There is an important difference between the way separate compilation can be implemented under this standard and the way it could be implemented under the previous standards. Under the previous standards, any information required to translate a program unit was specified in that program unit. Each translation was thus totally independent of all others. Under this standard, a program unit can use information that was specified in a separate module and thus may be dependent on that module. The implementation of this dependency in a processor may be that the translation of a program unit may depend on the results of translating one or more modules. Processors implementing the dependency this way are commonly described as supporting dependent compilation.

The dependencies involved here are new only in the sense that the Fortran processor is now aware of them. The same information dependencies existed under the previous standards, but it was the programmer's responsibility to transport the information necessary to resolve them by making redundant specifications of the information in multiple program units. The availability of separate but dependent compilation offers several potential advantages over the redundant textual specification of information:

(1) Specifying information at a single place in the program ensures that different program units using that information will be translated consistently. Redundant specification leaves the possibility that different information will erroneously be specified. Even if some kind of textual inclusion facility is used to ensure that the text of the specifications is identical in all involved program units, the presence of other specifications (for example, an IMPLICIT statement) may change the interpretation of that text.

(2) During the revision of a program, it is possible for a processor to assist in determining whether different program units have been translated using different (incompatible) versions of a module, although there is no requirement that a processor provide such assistance. Inconsistencies in redundant textual specification of information, on the other hand, tend to be much more difficult to detect.

(3) Putting information in a module provides a way of packaging it. Without modules, redundant specifications frequently must be interleaved with other specifications in a program unit, making convenient packaging of such information difficult.

(4) Because a processor may be implemented such that the specifications in a module are translated once and then repeatedly referenced, there is the potential for greater efficiency than when the processor must translate redundant specifications of information in multiple program units.

C.11.2.1 USE Statement and Dependent Compilation (11.3.2). Another benefit of the USE statement is its enhanced facilities for name management. If one needs to use only selected entities in a module, one can do so without having to worry about the names of all the other entities in that module. If one needs to use two different modules that happen to contain entities with the same name, there are several ways to deal with the conflict. If none of the entities with the same name are to be used, they can simply be ignored. If the name happens to refer to the same entity

in both modules (for example, if both modules obtained it from a third module), then there is no confusion about what the name denotes and the name can be freely used. If the entities are different and one or both is to be used, the local renaming facility in the USE statement makes it possible to give those entities different names in the program unit containing the USE statements.

A typical implementation of dependent but separate compilation may involve storing the result of translating a module in a file (or file element) whose name is derived from the name of the module. Note, however, that the name of a module is limited only by the Fortran rules and not by the names allowed in the file system. Thus the processor may have to provide a mapping between Fortran names and file system names.

See 3-7.10

The result of translating a module could reasonably either contain only the information textually specified in the module (with "pointers" to information originally textually specified in other modules) or contain all information specified in the module (including copies of information originally specified in other modules). Although the former approach would appear to save on storage space, the latter approach can greatly simplify the logic necessary to process a USE statement and can avoid the necessity of imposing a limit on the logical "nesting" of modules via the USE statement.

Variables declared in a module retain their definition status on much the same basis as variables in a common block. That is, saved variables retain their definition status throughout the execution of a program, while variables that are not saved retain their definition status only during the execution of scoping units that reference the module. In some cases, it may be appropriate to put a USE statement such as

```
USE MODULE, ONLY:
```

in a scoping unit in order to assure that other procedures that it references can communicate through the module. In such a case, the scoping unit would not access any entities from the module, but the variables not saved in the module would retain their definition status throughout the execution of the scoping unit.

There is an increased potential for undetected errors in a scoping unit that uses both implicit typing and the USE statement. For example, in the program fragment

```
SUBROUTINE SUB
   USE MY_MODULE
   IMPLICIT INTEGER (I-N), REAL (A-H, O-Z)
   X = F (B)
   A = G (X) + H (X + 1)
END SUBROUTINE
```

X could be either an implicitly typed real variable or a variable obtained from the module MY_MODULE and might change from one to the other because of changes in MY_MODULE unrelated to the action performed by SUB. Logic errors resulting from this kind of situation can be extremely difficult to locate. Thus, the use of these features together is discouraged.

C.11.2.2 Accessibility Attributes (11.3.1). The PUBLIC and PRIVATE attributes, which can be declared only in modules, can divide the entities in a module into those which are actually relevant to a scoping unit referencing the module and those that are not. This information may be used to improve the performance of a Fortran processor. For example, it may be possible to discard much of the information on the private entities once a module has been translated, thus saving on both storage and the time to search it. Similarly, it may be possible to recognize that two versions of a module differ only in the private entities they contain and avoid retranslating program units that use that module when switching from one version of the module to the other.

C.11.3 Pointers in Modules. A pointer from a module program unit may be accessible in a procedure via use association. Such pointers have a lifetime that is greater than targets that are declared in the procedure, unless such targets are saved. Therefore, if such a pointer is associated with a local target, there is the possibility that when the procedure completes execution, the target will cease to exist leaving the pointer "dangling". This standard considers such pointers to

be in an undefined state. They are neither associated nor disassociated. They must not be used again in the program until their status has been reestablished. There is no requirement on a processor to be able to detect when a pointer target ceases to exist.

C.11.4 Example of a Module (11.3). In addition to providing a portable means of avoiding the redundant specification of information in multiple program units, a module provides a convenient means of "packaging" related entities, such as the definitions of the representation and operations of an abstract data type. The following example of a module defines a data abstraction for a SET data type where the elements of each set are of type integer. The standard set operations of UNION, INTERSECTION, and DIFFERENCE are provided. The CARD function returns the cardinality of (number of elements in) its set argument. Two functions returning logical values are included, ELEMENT and SUBSET. ELEMENT overloads the operator .IN. and SUBSET overloads the operator <=. ELEMENT determines if a given scalar integer value is an element of a given set, and SUBSET determines if a given set is a subset of another given set. (Two sets may be checked for equality by comparing cardinality and checking that one is a subset of the other, or checking to see if each is a subset of the other.)

The transfer function SETF converts a vector of integer values to the corresponding set, with duplicate values removed. Thus, a vector of constant values can be used as set constants. An inverse transfer function VECTOR returns the elements of a set as a vector of values in ascending order. An assignment coercion allows assignment between sets of different sizes, and checks to see if the receiving set data object has an adequate maximum size (returning the null set if not). In this SET implementation, set data objects have a maximum size (number of elements in set) of 200.

```
MODULE INTEGER_SETS

INTEGER, PARAMETER :: MAX_SET_CARD = 200

TYPE SET                                    ! DEFINE SET DATA TYPE
   PRIVATE
   INTEGER CARDINAL_NUMBER
   INTEGER ELEMENT_VALUE (MAX_SET_CARD)       ! DEFINE SET DATA TYPE
END TYPE SET

INTERFACE OPERATOR (.IN.)
   MODULE PROCEDURE ELEMENT
END INTERFACE

INTERFACE OPERATOR (<=)
   MODULE PROCEDURE SUBSET
END INTERFACE

INTERFACE OPERATOR (+)
   MODULE PROCEDURE UNION
END INTERFACE

INTERFACE OPERATOR (-)
   MODULE PROCEDURE DIFFERENCE
END INTERFACE

INTERFACE OPERATOR (*)
   MODULE PROCEDURE INTERSECTION
END INTERFACE

INTEGER FUNCTION CARD (A)                  ! RETURNS CARDINALITY OF SET A
   TYPE (SET) A
   CARD = A % CARDINAL_NUMBER
```

```
END FUNCTION CARD

LOGICAL FUNCTION ELEMENT (X, A)                    ! DETERMINES IF
   INTEGER X                                       ! ELEMENT X IS IN SET A
   TYPE (SET) A
   ELEMENT = ANY (A % ELEMENT_VALUE (1 : A % CARDINAL_NUMBER) .EQ. X)
END FUNCTION ELEMENT

FUNCTION UNION (A, B)                              ! UNION OF SETS A AND B
   TYPE (SET) A, B, UNION
   INTEGER J
   UNION = A
   DO J = 1, B % CARDINAL_NUMBER
      IF (.NOT. (B % ELEMENT_VALUE (J) .IN. A)) THEN
         IF (UNION % CARDINAL_NUMBER < MAX_SET_CARD) THEN
            UNION % CARDINAL_NUMBER = UNION % CARDINAL_NUMBER + 1
            UNION % ELEMENT_VALUE (UNION % CARDINAL_NUMBER) = &
                  B % ELEMENT_VALUE (J)
         ELSE
            ! MAXIMUM SET SIZE EXCEEDED ....
         END IF
      END IF
   END DO
END FUNCTION UNION

FUNCTION DIFFERENCE (A, B)                 ! DIFFERENCE OF SETS A AND B
   TYPE (SET) A, B, DIFFERENCE
   INTEGER J, X
   DIFFERENCE % CARDINAL_NUMBER = 0        ! THE EMPTY SET
   DO J = 1, A % CARDINAL_NUMBER
      X = A % ELEMENT_VALUE (J)
      IF (.NOT. (X .IN. B)) DIFFERENCE = DIFFERENCE + SET (1, (/ X /))
   END DO
END FUNCTION DIFFERENCE

FUNCTION INTERSECTION (A, B)               ! INTERSECTION OF SETS A AND B
   TYPE (SET) A, B, INTERSECTION
   INTERSECTION = A - (A - B)
END FUNCTION INTERSECTION

LOGICAL FUNCTION SUBSET (A, B)                  ! DETERMINES IF SET A IS
   TYPE (SET) A, B                              ! A SUBSET OF SET B
   INTEGER I                                    ! OVERLOADS <= OPERATION
   SUBSET = A % CARDINAL_NUMBER <= B% CARDINAL_NUMBER
   IF (.NOT. SUBSET) RETURN                     ! FOR EFFICIENCY
   DO I = 1, A % CARDINAL_NUMBER
      SUBSET = SUBSET .AND. (A % ELEMENT_VALUE (I) .IN. B)
   END DO
END FUNCTION SUBSET

TYPE (SET) FUNCTION SETF (V)          ! TRANSFER FUNCTION BETWEEN A VECTOR
   INTEGER V (:)                      ! OF ELEMENTS AND A SET OF ELEMENTS
   INTEGER J                          ! REMOVING DUPLICATE ELEMENTS
   SETF % CARDINAL_NUMBER = 0
   DO J = 1, SIZE (V)
      IF (.NOT. (V (J) .IN. SETF)) THEN
         SETF % CARDINAL_NUMBER = SETF % CARDINAL_NUMBER + 1
```

```
            SETF % ELEMENT_VALUE (SETF % CARDINAL_NUMBER) = V (J)
         END IF
      END DO
   END FUNCTION SETF

   FUNCTION VECTOR (A)                    ! TRANSFER THE VALUES OF SET A
      TYPE (SET) A                        ! INTO A VECTOR OF ASCENDING ORDER
      INTEGER, POINTER :: VECTOR (:)
      INTEGER I, J, K
      ALLOCATE (VECTOR (A % CARDINAL_NUMBER))
      VECTOR = A % ELEMENT_VALUE (1 : A % CARDINAL_NUMBER)
      DO I = 1, A % CARDINAL_NUMBER - 1      ! USE A BETTER SORT IF
         DO J = I + 1, A % CARDINAL_NUMBER   ! A % CARDINAL_NUMBER IS LARGE
            IF (VECTOR (I) > VECTOR (J)) THEN
               K = VECTOR (J); VECTOR (J) = VECTOR (I); VECTOR (I) = K
            END IF
         END DO
      END DO
   END FUNCTION VECTOR

   END MODULE INTEGER_SETS
```

Examples of using INTEGER_SETS (A, B, and C are sets; X is an integer variable):

```
! CHECK TO SEE IF A HAS MORE THAN 10 ELEMENTS
IF (CARD (A) > 10) ...

! CHECK FOR X AN ELEMENT OF A BUT NOT OF B
IF (X .IN. (A - B)) ...

! C IS THE UNION OF A AND THE RESULT OF B INTERSECTED
! WITH THE INTEGERS 1 TO 100
C = A + B * SETF ((/ I, I = 1, 100 /))

! DOES A HAVE ANY EVEN NUMBERS IN THE RANGE 1:100?
IF (CARD (A * SETF ((/ I, I = 1, 200, 2 /)) > 0) ...

PRINT *, VECTOR (B) ! PRINT OUT THE ELEMENTS OF SET B, IN ASCENDING ORDER
```

C.12 Section 12 Notes.

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 insure that none of these names conflict with any of the names of the external procedures or other globally named entities in a standard-conforming program. For example, this might be done by including in each such added name a character that is not allowed in a standard-conforming name.

There is a potential portability problem in a scoping unit that references an external procedure without declaring it in either an EXTERNAL statement or a procedure interface block. On a different processor, the name of that procedure may be the name of a nonstandard intrinsic procedure and the processor would be permitted to interpret those procedure references as references to that intrinsic procedure. (On that processor, the program would also be viewed as not conforming to the standard because of the references to the nonstandard intrinsic procedure.) Declaration in an EXTERNAL statement or a procedure interface block causes the references to be to the external procedure regardless of the availability of an intrinsic procedure with the same

name. Note that declaration of the type of a procedure is not enough to make it external, even if the type is inconsistent with the type of the result of an intrinsic of the same name.

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).

Procedures defined by means other than Fortran are considered external procedures because their definitions are not contained within a Fortran program unit and because they are referenced using global names. The use of the term external should not be construed as any kind of restriction on the way in which these procedures may be defined. For example, if the means other than Fortran has its own facilities for internal and external procedures, it is permissible to use them. If the means other than Fortran can create an "internal" procedure with a global name, it is permissible for such an "internal" procedure to be considered by Fortran to be an external procedure. The means other than Fortran for defining external procedures, including any restrictions on the structure for organization of those procedures, are entirely processor dependent.

A Fortran processor may limit its support of procedures defined by means other than Fortran such that these procedures may affect entities in the Fortran environment only on the same basis as procedures written in Fortran. For example, it might prohibit the value of a local variable from being changed by a procedure reference unless that variable were one of the arguments to the procedure.

C.12.3 Procedure Interfaces (12.3). In FORTRAN 77, the interface to an external procedure was always deduced from the form of references to that procedure and any declarations of the procedure name in the referencing program unit. In this standard, features such as argument keywords and optional arguments make it impossible to deduce sufficient information about the dummy arguments from the nature of the actual arguments to be associated with them, and features such as array-valued function results and allocatable function results make necessary extensions to the declaration of a procedure that cannot be done in a way that would be analogous with the handling of such declarations in FORTRAN 77. Hence, mechanisms are provided through which all the information about a procedure's interface may be made available in a scoping unit that references it. A procedure whose interface must be deduced as in FORTRAN 77 is described as having an implicit interface. A procedure whose interface is fully known is described as having an explicit interface.

A scoping unit is allowed to contain a procedure interface block for procedures that do not exist in the executable program, provided the procedure described is never referenced. The purpose of this rule is to allow implementations in which the use of a module providing procedure interface blocks describing the interface of every routine in a library would not automatically cause each of those library routines to be a part of the program referencing the module. Instead, only those library procedures actually referenced would be a part of the executable program. (In implementation terms, the mere presence of a procedure interface block would not generate an external reference in such an implementation.)

C.12.4 Argument Association (12.4.1). There is a significant difference between the argument association allowed in this standard and that supported by FORTRAN 77 and FORTRAN 66. In FORTRAN 77 and 66, actual arguments were limited to consecutive storage units. With the exception of assumed length character dummy arguments, the structure imposed on that sequence of storage units was always determined in the invoked procedure and not taken from the actual argument. Thus it was possible to implement FORTRAN 66 and FORTRAN 77 argument association by supplying only the location of the first storage unit (except for character arguments, where the length would also have to be supplied). On the other hand, this standard allows arguments that do not reside in consecutive storage locations (for example, an array section), and dummy arguments that assume additional structural information from the actual argument (for example, assumed-shape dummy arguments). Thus, the mechanism to implement the argument

association allowed in this standard must be more general.

Because there are practical advantages to a processor that can support references to and from procedures defined by a FORTRAN 77 processor, requirements for explicit interfaces have been added to make it possible to determine whether a simple (FORTRAN 66/FORTRAN 77) argument association implementation mechanism is sufficient or whether the more general mechanism is necessary (12.3.1.1). Thus a processor can be implemented whose procedures expect the simple mechanism to be used whenever the procedure's interface is one which uses only FORTRAN 77 features and which expects the more general mechanism otherwise (for example, if there are assumed-shape or optional arguments). At the point of reference, the appropriate mechanism can be determined from the interface if it is explicit and can be assumed to be the simple mechanism if it is not. Note that if the simple mechanism is determined to be what the procedure expects, it may be necessary for the processor to allocate consecutive temporary storage for the actual argument, copy the actual argument to the temporary storage, reference the procedure using the temporary storage in place of the actual argument, copy the contents of temporary storage back to the actual argument, and deallocate the temporary storage.

Note that while this is the specific implementation method these rules were designed to support, it is not the only one possible. For example, on some processors, it may be possible to implement the general argument association in such a way that the information involved in FORTRAN 77 argument association may be found in the same places and the "extra" information is placed so it does not disturb a procedure expecting only FORTRAN 77 argument association. With such an implementation, argument association could be translated without regard to whether the interface is explicit or implicit. Alternatively, it would be possible to disallow discontiguous arguments when calling procedures defined by the FORTRAN 77 processor and let any copying to and from contiguous storage be done explicitly in the program. Yet another possibility would be not to allow references to procedures defined by a FORTRAN 77 processor.

C.12.5 Argument Intent Specification (12.4.1.1). Argument intent specifications serve several purposes in addition to documenting the intended use of dummy arguments. A processor can check whether an INTENT (IN) dummy argument is used in a way that could redefine it. A slightly more sophisticated processor could check to see whether an INTENT (OUT) dummy argument could possibly be referenced before it is defined. If the procedure's interface is explicit, the processor can also verify that actual arguments corresponding to INTENT (OUT) or INTENT (INOUT) dummy arguments are definable. A more sophisticated processor could use this information to optimize the translation of the referencing scoping unit by taking advantage of the fact that actual arguments corresponding to INTENT (IN) dummy arguments will not be changed and that any prior value of an actual argument corresponding to an INTENT (OUT) dummy argument will not be referenced and can thus be discarded.

Note that INTENT (OUT) means that the value of the argument after invoking the procedure is entirely the result of executing that procedure. If there is any possibility that an argument should retain its current value rather than being redefined, INTENT (INOUT) should be used rather than INTENT (OUT), even if there is no explicit reference to the value of the dummy argument.

Note also that INTENT (INOUT) is not equivalent to the default. The argument corresponding to an INTENT (INOUT) dummy argument always must be definable, while an argument corresponding to a dummy argument with default intent need be definable only if the dummy argument is actually redefined.

C.12.6 Dummy Argument Restrictions (12.5.2.9). The restrictions on entities associated with dummy arguments are intended to allow a processor to translate a procedure on the assumption that each dummy argument is distinct from any other entity accessible in the procedure. This allows a variety of optimizations in the translation of the procedure, including implementations of argument association in which the value of the actual argument is maintained in a register or in local storage.

C.12.7 Pointers as Arguments. If a dummy argument is declared to be a pointer, it may be matched only by an actual argument that also is a pointer, and the characteristics of both arguments must agree. A model for such an association is that descriptor values of the actual pointer are copied to the dummy pointer. If the actual pointer has an associated target, this target becomes accessible via the dummy pointer. If the dummy pointer becomes associated with a different target during execution of the procedure, this target will be accessible via the actual pointer after the procedure completes execution. If the dummy pointer becomes associated with a local target that ceases to exist when the procedure completes, the actual pointer will be left dangling in an undefined state. Such dangling pointers must not be used.

C.12.8 The ASSOCIATED Function. The ASSOCIATED intrinsic function may be used to test whether a pointer is associated with a target. The one-argument form is used for this purpose. In the two-argument form, the ASSOCIATED function tests whether the pointer first argument is associated with the space that is referred to by the second argument. The values in the two descriptors are compared. In most cases, it will be used to test if two pointers are associated with the same target.

what other reasons?

C.12.9 Internal Procedure Restrictions. This standard does not allow internal procedures to be used as actual arguments in part to simplify the problem of insuring that internal procedures with recursive hosts access entities from the correct instance of the host. If, as an extension, a processor allows internal procedures to be used as actual arguments, the correct instance in this case is the instance in which the procedure is supplied as an actual argument, even if the corresponding dummy argument is eventually invoked from a different instance.

NOT A PROBLEM

C.12.10 The Result Variable (12.5.2.2). The result variable is similar to any other variable local to a function subprogram. Its existence begins when execution of the function is initiated and ends when execution of the function is terminated. However, because the final value of this variable is used subsequently in the evaluation of the expression that invoked the function, an implementation may wish to defer releasing the storage occupied by that variable until after its value has been used in expression evaluation.

C.13 Section 13 Notes.

C.13.1 Summary of Features. This section is a summary of the principal array features.

C.13.1.1 Whole Array Expressions and Assignments (7.5.1.2, 7.5.1.5). An important new feature is that whole array expressions and assignments are permitted. For example, the statement

```
A = B + C * SIN (D)
```

where A, B, C, and D are arrays of the same shape, is permitted. It is interpreted element-by-element; that is, the sine function is taken on each element of D, each result is multiplied by the corresponding element of C, added to the corresponding element of B, and assigned to the corresponding element of A. Functions, including user-written functions, may be array valued and may overload scalar versions having the same name. All arrays in an expression or across an assignment must "conform"; that is, have exactly the same "shape" (number of dimensions and set of lengths in each dimension), but scalars may be included freely and these are interpreted as being broadcast to a conforming array. Expressions are evaluated before any assignment takes place.

C.13.1.2 Array Sections (2.4.7, 6.2.2.3). Whenever whole arrays may be used, it is also possible to use subarrays called "sections". For example:

```
A (:, 1:N, 2, 3:1:-1)
```

consists of a subarray containing the whole of the first dimension, positions 1 to N of the second dimension, position 2 of the third dimension and positions 1 to 3 in reverse order of the fourth dimension. This is an artificial example chosen to illustrate the different forms. Of course, the

most common use is to select a row or column of an array, for example:

```
A (:, J)
```

C.13.1.3 WHERE Statement (7.5.3). The WHERE statement applies a conforming logical array as a mask on the individual operations in the expression and in the assignment. For example:

```
WHERE (A .GT. 0) B = LOG (A)
```

takes the logarithm only for positive components of A and makes assignments only in these positions.

The WHERE statement also has a block form (WHERE construct).

C.13.1.4 Automatic and Allocatable Arrays (5.1, 5.1.2.4.3). A major advance for writing modular software is the presence of automatic arrays, created on entry to a subprogram and destroyed on return, and allocatable arrays whose rank is fixed but whose actual size and lifetime is fully under the programmer's control through explicit ALLOCATE and DEALLOCATE statements. The declarations

```
SUBROUTINE X (N, A, B)
   REAL WORK (N, N); REAL, ALLOCATABLE :: HEAP (:, :)
```

specify an automatic array WORK and an allocatable array HEAP. Note that a stack is an adequate storage mechanism for the implementation of automatic arrays, but a heap will be needed for allocatable arrays.

C.13.1.5 Array Constructors (4.5). Arrays, and in particular array constants, may be constructed with array constructors exemplified by:

```
(/ 1.0, 3.0, 7.2 /)
```

which is a rank-one array of size 3,

```
(/ (1.3, 2.7, L = 1, 10), 7.1 /)
```

which is a rank-one array of size 21 and contains (/ 1.3, 2.7 /) repeated 10 times followed by 7.1, and

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

which contains the integers 1, 2, ..., N. Only rank-one arrays may be constructed in this way, but higher dimensional arrays may be made from them by means of the intrinsic function RESHAPE.

C.13.1.6 Intrinsic Functions. All of the FORTRAN 77 intrinsic functions and all of the scalar intrinsic functions that have been added to the language have been extended to be applicable to arrays. Each such function is applied element-by-element to produce an array of the same shape. In addition, the following array intrinsics have been added, many of which return array-valued results.

C.13.1.6.1 Vector and Matrix Multiply Functions.

`DOTPRODUCT (VECTOR_A, VECTOR_B)`	Dot product of two arrays
`MATMUL (MATRIX_A, MATRIX_B)`	Matrix multiplication

C.13.1.6.2 Array Reduction Functions.

`ALL (ARRAY, DIM)`	True if all values are true
`ANY (ARRAY, DIM)`	True if any value is true
`COUNT (ARRAY, DIM)`	Number of true elements in an array.
`MAXVAL (ARRAY, DIM, MASK)`	Maximum value in an array
`MINVAL (ARRAY, DIM, MASK)`	Minimum value in an array
`PRODUCT (ARRAY, DIM, MASK)`	Product of array elements
`SUM (ARRAY, DIM, MASK)`	Sum of array elements

C.13.1.6.3 Array Inquiry Functions.

`ALLOCATED (ARRAY)`	Array allocation status
`LBOUND (ARRAY, DIM)`	Declared lower dimension bounds of an array
`SHAPE (SOURCE)`	Declared shape of an array or scalar
`SIZE (ARRAY, DIM)`	Declared total number of array elements
`UBOUND (ARRAY, DIM)`	Declared upper dimension bounds of an array

C.13.1.6.4 Array Construction Functions.

`MERGE (TSOURCE, FSOURCE, MASK)`	Merge under mask
`PACK (ARRAY, MASK, VECTOR)`	Pack an array into a vector under a mask
`RESHAPE (MOLD, SOURCE, PAD, ORDER)`	Reshape an array
`SPREAD (SOURCE, DIM, NCOPIES)`	Replicates an array by adding a dimension
`UNPACK (VECTOR, MASK, FIELD)`	Unpack a vector into an array under a mask

C.13.1.6.5 Array Manipulation Functions.

`CSHIFT (ARRAY, DIM, SHIFT)`	Circular shift
`EOSHIFT (ARRAY, DIM, SHIFT, BOUNDARY)`	End-off shift
`TRANSPOSE (MATRIX)`	Transpose of matrix

C.13.2 Examples. The array features have the potential to simplify the way that almost any array-using program is conceived and written. Many algorithms involving arrays can now be written conveniently as a series of computations with whole arrays.

C.13.2.1 Unconditional Array Computations. At the simplest level, statements such as A = B + C or S= SUM (A) can take the place of entire DO loops. The loops were required to perform array addition or to sum all the elements of an array.

Further examples of unconditional operations on arrays that are simple to write are:

matrix multiply	**P = MATMUL (Q, R)**
largest array element	**L = MAXVAL (P)**
factorial N	**F = PRODUCT ((/ (K, K = 2, N) /))**

The Fourier sum $F = \sum_{i=1}^{N} a_i \times \cos x_i$ may also be computed without writing a DO loop if one makes use of the element-by-element definition of array expressions as described in Section 7. Thus, we can write

```
F = SUM (A * COS (X)).
```

The successive stages of calculation of F would then involve the arrays:

A	=	**(/ A(1), ..., A(N) /)**
X	=	**(/ X(1), ..., X(N) /)**
COS(X)	=	**(/ COS(X(1)), ..., COS(X(N)) /)**
A*COS(X)	=	**(/ A(1) * COS(X(1)), ..., A(N) * COS(X(N)) /)**

The final scalar result is obtained simply by summing the elements of the last of these arrays. Thus, the processor is dealing with arrays at every step of the calculation.

C.13.2.2 Conditional Array Computations. Suppose we wish to compute the Fourier sum in the above example, but to include only those terms $a(i)\cos x(i)$ that satisfy the condition that the coefficient $a(i)$ is less than 0.01 in absolute value. More precisely, we are now interested in evaluating the conditional Fourier sum

$$CF = \sum_{|a_i| < 0.01} a_i \times \cos x_i$$

where the index runs from 1 to N as before.

This can be done using the MASK parameter of the SUM function, which restricts the summation of the elements of the array A * COS(X) to those elements that correspond to true elements of MASK. Clearly, the mask required is the logical array expression ABS(A) .LT. 0.01. Note that the stages of evaluation of this expression are:

A	=	(/ A(1), ..., A(N) /)
ABS(A)	=	(/ ABS(A(1)), ..., ABS(A(N)) /)
ABS(A) .LT. 0.01	=	(/ ABS(A(1) .LT. 0.01, ..., ABS(A(N)) .LT. 0.01 /)

The conditional Fourier sum we arrive at is:

```
CF = SUM (A * COS (X), MASK = ABS (A) .LT. 0.01)
```

If the mask is all false, the value of CF is zero.

The use of a mask to define a subset of an array is crucial to the action of the WHERE statement. Thus for example, to set an entire array to zero, we may write simply A = 0; but to set only the negative elements to zero, we need to write the conditional assignment

```
WHERE (A .LT. 0)  A = 0
```

The WHERE statement complements ordinary array assignment by providing array assignment to any subset of an array that can be restricted by a logical expression.

In the Ising model described below, the WHERE statement predominates in use over the ordinary array assignment statement.

C.13.2.3 A Simple Program: The Ising Model. The Ising model is a well-known Monte Carlo simulation in 3-dimensional Euclidean space which is useful in certain physical studies. We will consider in some detail how this might be programmed. The model may be described in terms of a logical array of shape N by N by N. Each gridpoint is a single logical variable which is to be interpreted as either an up-spin (true) or a down-spin (false).

The Ising model operates by passing through many successive states. The transition to the next state is governed by a local probabilistic process. At each transition, all gridpoints change state simultaneously. Every spin either flips to its opposite state or not according to a rule that depends only on the states of its 6 nearest neighbors in the surrounding grid. The neighbors of gridpoints on the boundary faces of the model cube are defined by assuming cubic periodicity. In effect, this extends the grid periodically by replicating it in all directions throughout space.

The rule states that a spin is flipped to its opposite parity for certain gridpoints where a mere 3 or fewer of the 6 nearest neighbors currently have the same parity as it does. Also, the flip is executed only with probability P (4), P (5), or P (6) if as many as 4, 5, or 6 of them have the same parity as it does. (The rule seems to promote neighborhood alignments that may presumably lead to equilibrium in the long run).

C.13.2.3.1 Problems To Be Solved. Some of the programming problems that we will need to solve in order to translate the Ising model into Fortran statements using entire arrays are:

(1) Counting nearest neighbors that have the same spin;

(2) Providing an array-valued function to return an array of random numbers; and

(3) Determining which gridpoints are to be flipped.

C.13.2.3.2 Solutions in Fortran. The arrays needed are:

```
LOGICAL ISING (N, N, N), FLIPS (N, N, N)
INTEGER ONES (N, N, N), COUNT (N, N, N)
REAL THRESHOLD (N, N, N)
```

The array-valued function needed is:

```
FUNCTION RAND (N)
REAL RAND (N, N, N)
```

The transition probabilities are specified in the array

```
REAL P (6)
```

The first task is to count the number of nearest neighbors of each gridpoint *g* that have the same spin as *g*.

Assuming that ISING is given to us, the statements

```
ONES = 0
WHERE (ISING) ONES = 1
```

make the array ONES into an exact analog of ISING in which 1 stands for an up-spin and 0 for a down-spin.

The next array we construct, COUNT, will record for every gridpoint of ISING the number of spins to be found among the 6 nearest neighbors of that gridpoint. COUNT will be computed by adding together 6 arrays, one for each of the 6 relative positions in which a nearest neighbor is found. Each of the 6 arrays is obtained from the ONES array by shifting the ONES array one place circularly along one of its dimensions. This use of circular shifting imparts the cubic periodicity.

```
COUNT = CSHIFT(ONES, DIM = 1, SHIFT = -1)  &
       +CSHIFT(ONES, DIM = 1, SHIFT =  1)  &
       +CSHIFT(ONES, DIM = 2, SHIFT = -1)  &
       +CSHIFT(ONES, DIM = 2, SHIFT =  1)  &
       +CSHIFT(ONES, DIM = 3, SHIFT = -1)  &
       +CSHIFT(ONES, DIM = 3, SHIFT =  1)
```

At this point, COUNT contains the count of nearest neighbor up-spins even at the gridpoints where the Ising model has a down-spin. But we want a count of down-spins at those gridpoints, so we correct COUNT at the down (false) points of ISING by writing:

```
WHERE (.NOT. ISING)  COUNT = 6 - COUNT
```

Our object now is to use these counts of what may be called the "like-minded nearest neighbors" to decide which gridpoints are to be flipped. This decision will be recorded as the true elements of an array FLIP. The decision to flip will be based on the use of uniformly distributed random numbers from the interval $0 \le p < 1$. These will be provided at each gridpoint by the array-valued function RAND. The flip will occur at a given point if and only if the random number at that point is less than a certain threshold value. In particular, by making the threshold value equal to 1 at the points where there are 3 or fewer like-minded nearest neighbors, we guarantee that a flip occurs at those points (because p is always less than 1). Similarly, the threshold values corresponding to counts of 4, 5, and 6 are set to P (4), P (5), and P (6) in order to achieve the desired probabilities of a flip at those points (P (4), P (5), and P (6) are input parameters in the range 0 to 1).

The thresholds are established by the statements:

```
THRESHOLD = 1.0
WHERE (COUNT .EQ. 4) THRESHOLD = P (4)
WHERE (COUNT .EQ. 5) THRESHOLD = P (5)
WHERE (COUNT .EQ. 6) THRESHOLD = P (6)
```

and the spins that are to be flipped are located by the statement:

```
FLIPS = RAND (N) .LE. THRESHOLD
```

All that remains to complete one transition to the next state of the ISING model is to reverse the spins in ISING wherever FLIPS is true:

```
WHERE (FLIPS)  ISING = .NOT. ISING
```

C.13.2.3.3 The Complete Fortran Subroutine. The complete code, enclosed in a subroutine that performs a sequence of transitions, is as follows:

```
SUBROUTINE TRANSITION (N, ISING, ITERATIONS, P)

   LOGICAL ISING (N, N, N), FLIPS (N, N, N)
   INTEGER ONES (N, N, N), COUNT (N, N, N)
   REAL THRESHOLD (N, N, N), P (6)

   DO I = 1, ITERATIONS
      ONES = 0
      WHERE (ISING)  ONES = 1
      COUNT = CSHIFT (ONES, 1, -1) + CSHIFT (ONES, 1, 1) &
             +CSHIFT (ONES, 2, -1) + CSHIFT (ONES, 2, 1) &
             +CSHIFT (ONES, 3, -1) + CSHIFT (ONES, 3, 1)
      WHERE (.NOT. ISING)  COUNT = 6 - COUNT
      THRESHOLD = 1.0
      WHERE (COUNT .EQ. 4)  THRESHOLD = P (4)
      WHERE (COUNT .EQ. 5)  THRESHOLD = P (5)
      WHERE (COUNT .EQ. 6)  THRESHOLD = P (6)
      FLIPS = RAND (N) .LE. THRESHOLD
      WHERE (FLIPS)  ISING = .NOT. ISING
   END DO

CONTAINS
   FUNCTION RAND (N)
      REAL RAND (N, N, N)
      CALL RANDOM (HARVEST = RAND)
      RETURN
   END FUNCTION RAND
END
```

An "element-by-element" subscript would be useful:
Threshold = P[count]
with P(1:3) = 1.0

C.13.2.3.4 Reduction of Storage. The array ISING could be removed (at some loss of clarity) by representing the model in ONES all the time. The array FLIPS can be avoided by combining the two statements that use it as:

```
WHERE (RAND (N) .LE. THRESHOLD)  ISING = .NOT. ISING
```

but an extra temporary array would probably be needed. Thus, the scope for saving storage while performing whole array operations is limited. If N is small, this will not matter and the use of whole array operations is likely to lead to good execution speed. If N is large, storage may be very important and adequate efficiency will probably be available by performing the operations plane by plane. The resulting code is not as elegant, but all the arrays except ISING will have size of order N^2 instead of N^3.

C.13.3 FORmula TRANslation and Array Processing. Many mathematical formulas can be translated directly into Fortran by use of the array processing features.

We assume the following array declarations:

```
REAL X (N), A (M, N)
```

Some examples of mathematical formulas and corresponding Fortran expressions follow.

C.13.3.1 A Sum of Products. The expression

$$\sum_{j=1}^{N} \prod_{i=1}^{M} a_{ij}$$

can be formed using the Fortran expression

```
SUM (PRODUCT (A, DIM=1))
```

The argument DIM=1 means that the product is to be computed down each column of A. If A had the value

$$\begin{bmatrix} B & C & D \\ E & F & G \end{bmatrix}$$

the result of this expression is BE + CF + DG.

C.13.3.2 A Product of Sums. The expression

$$\prod_{i=1}^{M} \sum_{j=1}^{N} a_{ij}$$

can be formed using the Fortran expression

```
PRODUCT (SUM (A, DIM = 2))
```

The argument DIM = 2 means that the sum is to be computed along each row of A. If A had the value

$$\begin{bmatrix} B & C & D \\ E & F & G \end{bmatrix}$$

the result of this expression is (B+C+D)(E+F+G).

C.13.3.3 Addition of Selected Elements. The expression

$$\sum_{x_i > 0.0} x_i$$

can be formed using the Fortran expression

```
SUM (X, MASK = X .GT. 0.0)
```

The mask locates the positive elements of the array of rank one. If X has the vector value (0.0, –0.1, 0.2, 0.3, 0.2, –0.1, 0.0) the result of this expression is 0.7.

C.13.4 Sum of Squared Residuals. The expression

$$\sum_{i=1}^{N} (x_i - x_{mean})^2$$

can be formed using the Fortran statements

```
XMEAN = SUM (X) / SIZE (X)
SS = SUM ((X - XMEAN) ** 2)
```

Thus, SS is the sum of the squared residuals.

C.13.5 Vector Norms: Infinity-Norm and One-Norm. The infinity-norm of vector X = (X(1), ..., X(N)) is defined as the largest of the numbers ABS (X(1)), ..., ABS (X(N)) and therefore has the value MAXVAL (ABS (X)).

The one-norm of vector X is defined as the *sum* of the numbers ABS (X(1)), ..., ABS (X(N)) and therefore has the value SUM (ABS (X)).

C.13.6 Matrix Norms: Infinity-Norm and One-Norm. The infinity-norm of the matrix A = (A (I, J)) is the largest row-sum of the matrix ABS (A (I, J)) and therefore has the value MAXVAL (SUM (ABS (A), DIM = 2)).

The one-norm of the matrix A = (A (I, J)) is the largest column-sum of the matrix ABS (A (I, J)) and therefore has the value MAXVAL (SUM (ABS (A), DIM = 1)).

C.13.7 Logical Queries. The intrinsic functions allow quite complicated questions about tabular data to be answered without use of loops or conditional constructs. Consider, for example, the questions asked below about a simple tabulation of students' test scores.

Suppose the rectangular table T (M, N) contains the test scores of M students who have taken N different tests. T is an integer matrix with entries in the range 0 to 100.

Example: The scores on 4 tests made by 3 students are held as the table

$$T = \begin{bmatrix} 85 & 76 & 90 & 60 \\ 71 & 45 & 50 & 80 \\ 66 & 45 & 21 & 55 \end{bmatrix}$$

Question: What is each student's top score?

Answer: MAXVAL (T, DIM = 2); in the example: [90, 80, 66].

Question: What is the average of all the scores?

Answer: SUM (T) / SIZE (T); in the example: 62.

Question: How many of the scores in the table are above average?

Answer: ABOVE = T .GT. SUM (T) / SIZE (T); N = COUNT (ABOVE); in the example: ABOVE is the logical array (t = true, . = false):

$$\begin{bmatrix} t & t & t & . \\ t & . & . & t \\ t & . & . & . \end{bmatrix}$$

and COUNT (ABOVE) is 6.

Question: What was the lowest score in the above-average group of scores?

Answer: MINVAL (T, MASK = ABOVE), where ABOVE is as defined previously; in the example: 66.

Question: Was there a student whose scores were all above average?

Answer: With ABOVE as previously defined, the answer is yes or no according as the value of the expression ANY (ALL (ABOVE, DIM = 2)) is true or false; in the example, the answer is no.

C.13.8 Parallel Computations. The most straightforward kind of parallel processing is to do the same thing at the same time to many operands. Matrix addition is a good example of this very simple form of parallel processing. Thus, the array assignment A = B + C specifies that corresponding elements of the identically-shaped arrays B and C be added together in parallel and that the resulting sums be assigned in parallel to the array A.

The "process" being done "in parallel" in the example of matrix addition is of course the process of addition. And the array feature that so successfully implements matrix addition as a parallel process is the element-by-element evaluation of array expressions.

These observations lead us to look to element-by-element computation as a means of implementing other simple parallel processing algorithms.

C.13.9 Example of Element-by-Element Computation. Several polynomials of the same degree may be evaluated at the same point by arrangeing their coefficients as the rows of a matrix and applying Horner's method for polynomial evaluation to the columns of the matrix so formed.

The procedure is illustrated by the code to evaluate the three cubic polynomials:

$$P(t) = 1+2t-3t^2+4t^3$$

$$Q(t) = 2-3t+4t^2-5t^3$$

$$R(t) = 3+4t-5t^2+6t^3$$

in parallel at the point t = X and to place the resulting vector of numbers [P(X), Q(X), R(X)] in the real array RESULT (3).

The code to compute RESULT is just the one statement

```
RESULT = M (:, 1) + X * (M (:, 2) + X * (M (:, 3) + X * M (:, 4)))
```

where M represents the matrix M (3, 4) with value

$$\begin{bmatrix} 1 & 2 & -3 & 4 \\ 2 & -3 & 4 & -5 \\ 3 & 4 & -5 & 6 \end{bmatrix}$$

C.14 Section 14 Notes.

C.14.1 Storage Association of Zero-Sized Objects. Zero-sized objects may occur in a storage association context as the result of changing a parameter. For example, a program might contain the following declarations:

```
INTEGER, PARAMETER :: PROBSIZE = 10
INTEGER, PARAMETER :: ARRAYSIZE = PROBSIZE * 100
REAL, DIMENSION (ARRAYSIZE) :: X
INTEGER, DIMENSION (ARRAYSIZE) :: IX
...
COMMON / EXAMPLE / A, B, C, X, Y, Z
EQUIVALENCE (X, IX)
...
```

If the first statement is subsequently changed to set PROBSIZE to zero, the program still will conform to the standard.

APPENDIX D. SYNTAX RULES

(This appendix is not part of American National Standard X3.9-198x, but is included for information only.)

1. INTRODUCTION

2. FORTRAN TERMS AND CONCEPTS

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

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

R1101 *main-program* **is** [*program-stmt*]
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-program-stmt

R1217 *external-subprogram* **is** *procedure-heading*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
procedure-ending

R1218 *procedure-heading* **is** *function-stmt*
or *subroutine-stmt*

R1220 *procedure-ending* **is** *end-function-stmt*
or *end-subroutine-stmt*

R1104 *module* **is** *module-stmt*
[*specification-part*]
[*module-subprogram-part*]
end-module-stmt

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

R203 *specification-part* **is** [*use-stmt*] ...
[*implicit-part*]
[*declaration-construct*] ...

R204 *implicit-part* **is** [*implicit-part-stmt*] ...
implicit-stmt

R205 *implicit-part-stmt* **is** *implicit-stmt*
or *parameter-stmt*
or *format-stmt*
or *entry-stmt*

R206 *declaration-construct* **is** *derived-type-def*
or *interface-block*
or *type-declaration-stmt*
or *specification-stmt*
or *parameter-stmt*
or *format-stmt*

or *entry-stmt*
or *stmt-function-stmt*

R207 *execution-part* **is** *executable-construct*
[*execution-part-construct*] ...

R208 *execution-part-construct* **is** *executable-construct*
or *format-stmt*
or *data-stmt*
or *entry-stmt*

R209 *internal-subprogram-part* **is** *contains-stmt*
internal-subprogram
[*internal-subprogram*] ...

R210 *internal-subprogram* **is** *procedure-heading*
[*specification-part*]
[*execution-part*]
procedure-ending

R211 *module-subprogram-part* **is** *contains-stmt*
module-subprogram
[*module-subprogram*] ...

R212 *module-subprogram* **is** *procedure-heading*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
procedure-ending

R213 *specification-stmt* **is** *access-stmt*
or *common-stmt*
or *data-stmt*
or *dimension-stmt*
or *equivalence-stmt*
or *external-stmt*
or *intent-stmt*
or *intrinsic-stmt*
or *namelist-stmt*
or *optional-stmt*
or *save-stmt*

R214 *executable-construct* **is** *action-stmt*
or *case-construct*
or *do-construct*
or *if-construct*
or *where-construct*

R215 *action-stmt* **is** *allocate-stmt*
or *assignment-stmt*
or *backspace-stmt*
or *call-stmt*
or *close-stmt*
or *computed-goto-stmt*
or *continue-stmt*
or *cycle-stmt*
or *deallocate-stmt*
or *endfile-stmt*
or *end-function-stmt*
or *end-program-stmt*
or *end-subroutine-stmt*
or *exit-stmt*

or *goto-stmt*
or *if-stmt*
or *inquire-stmt*
or *nullify-stmt*
or *open-stmt*
or *pointer-assignment-stmt*
or *print-stmt*
or *read-stmt*
or *return-stmt*
or *rewind-stmt*
or *stop-stmt*
or *where-stmt*
or *write-stmt*
or arithmetic-if-stmt
or assign-stmt
or assigned-goto-stmt
or pause-stmt

3. CHARACTERS, LEXICAL TOKENS, AND SOURCE FORM

R316 *character* **is** *alphanumeric-character*
or *special-character*

R317 *alphanumeric-character* **is** *letter*
or *digit*
or *underscore*

R318 *underscore* **is** _

R319 *name* **is** *letter* [*alphanumeric-character*]...

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

R320 *constant* **is** *literal-constant*
or *named-constant*

R321 *literal-constant* **is** *int-literal-constant*
or *real-literal-constant*
or *complex-literal-constant*
or *logical-literal-constant*
or *char-literal-constant*
or *boz-literal-constant*

R322 *named-constant* **is** *name*

R323 *int-constant* **is** *constant*

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

R324 *char-constant* **is** *constant*

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

R325 *intrinsic-operator* **is** *power-op*
or *mult-op*
or *add-op*
or *concat-op*
or *rel-op*
or *not-op*
or *and-op*
or *or-op*
or *equiv-op*

R708 *power-op* **is** **
R709 *mult-op* **is** *
or /
R710 *add-op* **is** +
or -
R712 *concat-op* **is** //
R714 *rel-op* **is** .EQ.
or .NE.
or .LT.
or .LE.
or .GT.
or .GE.
or ==
or <>
or <
or <=
or >
or >=
R719 *not-op* **is** .NOT.
R720 *and-op* **is** .AND.
R721 *or-op* **is** .OR.
R722 *equiv-op* **is** .EQV.
or .NEQV.
R326 *defined-operator* **is** *defined-unary-op*
or *defined-binary-op*
or *overloaded-intrinsic-op*
R704 *defined-unary-op* **is** . *letter* [*letter*]... .
R724 *defined-binary-op* **is** . *letter* [*letter*]... .
R327 *overloaded-intrinsic-op* **is** *intrinsic-operator*

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

R328 *label* **is** *digit* [*digit* [*digit* [*digit* [*digit*]]]]

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

4. INTRINSIC AND DERIVED DATA TYPES

R429 *signed-digit-string* **is** [*sign*] *digit-string*
R430 *digit-string* **is** *digit* [*digit*]...
R431 *signed-int-literal-constant* **is** [*sign*] *int-literal-constant*
R432 *int-literal-constant* **is** *digit-string* [_*kind-param*]
R433 *kind-param* **is** [*sign*] *digit-string*
or *scalar-int-constant-name*
R434 *sign* **is** +
or -
R435 *boz-literal-constant* **is** *binary-constant*

or *octal-constant*
or *hex-constant*

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

R436 *binary-constant* **is** B' *digit* [*digit*] ... '
or B" *digit* [*digit*] ... "

Constraint: *digit* may have only the values 0 or 1.

R437 *octal-constant* **is** O' *digit* [*digit*] ... '
or O" *digit* [*digit*] ... "

Constraint: *digit* may have only the values 0 through 7.

R438 *hex-constant* **is** Z' *hex-digit* [*hex-digit*] ... '
or Z" *hex-digit* [*hex-digit*] ... "

R439 *hex-digit* **is** digit
or A
or B
or C
or D
or E
or F

R440 *signed-real-literal-constant* **is** [*sign*] *real-literal-constant*

R441 *real-literal-constant* **is** *significand*
[*exponent-letter exponent*] [_ *kind-param*]
or *digit-string*
exponent-letter exponent [_ *kind-param*]

R442 *significand* **is** *digit-string* .
[*digit-string*]
or . *digit-string*

R443 *exponent* **is** *signed-digit-string*

R444 *exponent-letter* **is** E
or D

R445 *complex-literal-constant* **is** (*real-part* , *imag-part*)

R446 *real-part* **is** *signed-int-literal-constant*
or *signed-real-literal-constant*

R447 *imag-part* **is** *signed-int-literal-constant*
or *signed-real-literal-constant*

R448 *char-literal-constant* **is** ' [*rep-char*] ... ' [_ *kind-param*]
or " [*rep-char*] ... " [_ *kind-param*]

R449 *logical-literal-constant* **is** .TRUE. [_ *kind-param*]
or .FALSE. [_ *kind-param*]

R450 *derived-type-def* **is** *derived-type-stmt*
[PRIVATE]
[SEQUENCE]
component-def-stmt
[*component-def-stmt*] ...
end-type-stmt

R451 *derived-type-stmt* **is** [*access-spec*] TYPE *type-name*

Constraint: If SEQUENCE is present, all derived types used in component definitions must also be SEQUENCE structures.

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

R452 *end-type-stmt* **is** END TYPE [*type-name*]

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

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

R453 *component-def-stmt* **is** *type-spec* [[, *component-attr-spec-list*] ::] ■
■ *component-decl-list*

R454 *component-attr-~~stmt~~ spec* **is** POINTER
or DIMENSION (*component-array-spec*)

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

Constraint: A *type-spec* (5.1) in a *component-def-stmt* may include the *type-name* of its containing *derived-type-def* only if the POINTER attribute is specified for that component.

R455 *component-array-spec* **is** *explicit-shape-spec-list*
or *deferred-shape-spec-list*

R456 *component-decl* **is** *component-name [(component-array-spec)]* ■
■ *[* char-length]*

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

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

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

Constraint: A *char-length* in a *component-decl* must be an integer constant expression.

Constraint: Each bound in the *explicit-shape-spec* (R425 and R426) must be an integer constant expression.

R457 *structure-constructor* **is** *type-name (expr-list)*

R458 *array-constructor* **is** *(/ ac-value-list /)*

R459 *ac-value* **is** *expr*
or *ac-implied-do*

R460 *ac-implied-do* **is** *(ac-value-list , ac-implied-do-control)*

R461 *ac-implied-do-control* **is** *ac-do-variable = scalar-int-expr* , ■
■ *scalar-int-expr [, scalar-int-expr]*

R462 *ac-do-variable* **is** *scalar-int-variable*

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

5. DATA OBJECT DECLARATIONS AND SPECIFICATIONS

R563 *type-declaration-stmt* **is** *type-spec [[, attr-spec]... ::] entity-decl-list*

R564 *type-spec* **is** *INTEGER [kind-selector]*
or *REAL [kind-selector]*
or *DOUBLE PRECISION*
or *COMPLEX [kind-selector]*
or *CHARACTER [char-selector]*
or *LOGICAL [kind-selector]*

or *TYPE (type-name)* .

R565 attr-spec **is** *PARAMETER*
or *access-spec*
or *ALLOCATABLE*
or *DIMENSION (array-spec)*
or *EXTERNAL*
or *INTENT (intent-spec)*
or *INTRINSIC*
or *OPTIONAL*
or *POINTER*
or *SAVE*
or *TARGET*

R566 entity-decl **is** *object-name [(array-spec)]* ■
■ *[* char-length] [= restricted-constant-expr]*
or *function-name [(array-spec)] [* char-length]*

R567 kind-selector **is** *([KIND =] scalar-int-restricted-constant-expr)*

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

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

Constraint: The = *restricted-constant-expr* must appear if the statement contains a *PARAMETER* attribute (5.1.2.1).

Constraint: If = *restricted-constant-expr* appears, :: must appear before the *entity-decl-list*.

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

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

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

Constraint: An object must not have both the TARGET attribute and the PARAMETER attribute.

Constraint: If the POINTER attribute is specified, INTENT must not be specified.

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

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

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

Constraint: The SAVE attribute must not be specified for an object that is in a common block, a dummy argument, a procedure, a function result, or an automatic data object.

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

Constraint: An entity must not have the EXTERNAL or INTRINSIC attribute unless it is a function.

R568 char-selector **is** *length-selector*
or *([LEN=] type-param-value , ■*
■ *[KIND=] scalar-int-restricted-constant-expr)*
or *(KIND= scalar-int-restricted-constant-expr ■*
■ *[, LEN= type-param-value])*

R569 length-selector **is** *([LEN =] type-param-value)*
or * *char-length [,]*

R570 char-length **is** *(type-param-value)*
or *scalar-int-literal-constant*

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

R571 *type-param-value* **is** *specification-expr*
or *

R572 *access-spec* **is** *PUBLIC*
or *PRIVATE*

R573 *intent-spec* **is** *IN*
or *OUT*
or *INOUT*

R574 *array-spec* **is** *explicit-shape-spec-list*
or *assumed-shape-spec-list*
or *deferred-shape-spec-list*
or *assumed-size-spec*

Constraint: The maximum rank is seven.

R575 *explicit-shape-spec* **is** *[lower-bound :] upper-bound*

R576 *lower-bound* **is** *scalar-int-expr*

R577 *upper-bound* **is** *scalar-int-expr*

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

Constraint: The bounds in an explicit-shape array declaration must be specification expressions (7.1.6.2).

R578 *assumed-shape-spec* **is** *[lower-bound] :*

R579 *deferred-shape-spec* **is** *:*

R580 *assumed-size-spec* **is** *[explicit-shape-spec-list ,] [lower-bound :] **

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

R581 *intent-stmt* **is** *INTENT (intent-spec) [::] dummy-arg-name-list*

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

R582 *optional-stmt* **is** *OPTIONAL [::] dummy-arg-name-list*

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

R583 *access-stmt* **is** *access-spec [[::] use-name-list]*

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

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

Constraint: A *use-name* in a PUBLIC statement must not be the name of a module procedure that has a dummy argument of PRIVATE type.

R584 *save-stmt* **is** *SAVE [[::] saved-entity-list]*

R585 *saved-entity* **is** *name*
or */ common-block-name /*

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

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

R586 *dimension-stmt* **is** *DIMENSION array-name (array-spec)*
[, array-name (array-spec)]...

R587 *data-stmt* **is** *DATA data-stmt-set [[,] data-stmt-set]...*

R588 *data-stmt-set* **is** *data-stmt-object-list / data-stmt-value-list /*

R589 *data-stmt-object* **is** *variable*
or *data-implied-do*

Constraint: The *data-stmt-object* must not be a constant.

R590 *data-stmt-value* **is** *[data-stmt-repeat *] data-stmt-constant*

R591 *data-stmt-constant* **is** *scalar-constant*
or *signed-int-literal-constant*
or *signed-real-literal-constant*
or *structure-constructor*
or *unsigned-boz-literal-constant*

R592 *data-stmt-repeat* **is** *scalar-int-constant*

R593 *data-implied-do* **is** *(data-i-do-object-list, data-i-do-variable =*
scalar-int-expr, scalar-int-expr [, scalar-int-expr])

R594 *data-i-do-object* **is** *array-element*
or *data-implied-do*

R595 *data-i-do-variable* **is** *scalar-int-variable*

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

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

Constraint: If a *data-stmt-constant* is a *structure-constructor*, each component must be a constant expression.

Constraint: A variable whose name is included in a *data-stmt-object-list* or a *data-i-do-object-list* must not be a dummy argument, made accessible by use association or host association, in a named common block unless the DATA statement is in a block data program unit, in a blank common block, a character string with zero length, a function name, an automatic object, a pointer, an allocatable array, or a zero-sized array.

Constraint: A subscript in an array element *data-i-do-object* must be an expression whose primaries are either constants or DO variables of the containing *data-implied-dos*. Each such DO variable must appear in some subscript of the *array-element*.

Constraint: A *scalar-int-expr* of a *data-implied-do* must involve as primaries only constants or DO variables of the containing *data-implied-dos*.

R596 *parameter-stmt* **is** *PARAMETER (named-constant-def-list)*

R597 *named-constant-def* **is** *named-constant = restricted-constant-expr*

R598 *implicit-stmt* **is** *IMPLICIT implicit-spec-list*
or *IMPLICIT NONE*

R599 *implicit-spec* **is** *type-spec (letter-spec-list)*

R5100 *letter-spec* **is** *letter [– letter]*

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

R5101 namelist-stmt **is** *NAMELIST / namelist-group-name / namelist-group-object-list*
[*[,] / namelist-group-name / namelist-group-object-list*]...

R5102 namelist-group-object **is** *variable-name*

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

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

R5103 equivalence-stmt **is** *EQUIVALENCE equivalence-set-list*

R5104 equivalence-set **is** *(equivalence-object , equivalence-object-list)*

R5105 equivalence-object **is** *variable-name*
or *array-element*
or *substring*

Constraint: An *equivalence-object* must not be a dummy argument, a pointer, an allocatable array, an automatic object, or a subobject of any such object, or a function name.

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

Constraint: If an *equivalence-object* is of derived type, all of the objects in the list must be of the same derived type; or all of the objects must be default numeric of default logical type or be of derived type with numeric storage association; or all of the objects must be default character type or be of derived type with character storage association. Otherwise, the results are processor dependent.

Constraint: If an equivalence object is an object of intrinsic type with nondefault kind parameters, all of the objects must be of the same type or the result is processor dependent.

R5106 common-stmt **is** *COMMON [/ [common-block-name] /]*
common-block-object-list
[[,] / [common-block-name] /
common-block-object-list]...

R5107 common-block-object **is** *variable-name [(explicit-shape-spec-list)]*

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

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

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

Constraint: If an object of derived type appears in a common block, it must be a sequenced structure or a storage-associated structure.

6. USE OF DATA OBJECTS

R6108 variable **is** *scalar-variable-name*
or *array-variable-name*
or *subobject*

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

R6109 subobject **is** *array-element*

or *array-section*
or *structure-component*
or *substring*

R6110 logical-variable **is** *variable*

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

R6111 char-variable **is** *variable*

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

R6112 int-variable **is** *variable*

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

R6113 substring **is** *parent-string (substring-range)*

R6114 parent-string **is** *scalar-variable-name*
or *array-element*
or *scalar-structure-component*
or *scalar-constant*

R6115 substring-range **is** *[scalar-int-expr] : [scalar-int-expr]*

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

R6116 structure-component **is** *parent-structure % component-name*

R6117 parent-structure **is** *scalar-variable-name*
or *array-variable-name*
or *array-element*
or *array-section*
or *structure-component*
or *named-constant*

Constraint: If *parent-structure* is an array, the component must not be an array.

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

Constraint: *component-name* must be a component from the derived-type definition of the type of *parent-structure*.

R6118 array-element **is** *parent-array (subscript-list)*

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

R6119 array-section **is** *parent-array (section-subscript-list) [(substring-range)]*

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

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

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

R6120 parent-array **is** *array-name*
or *structure-component*

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

R6121 subscript **is** *scalar-int-expr*

R6122 section-subscript **is** *subscript*
or *subscript-triplet*
or *vector-subscript*

R6123 subscript-triplet **is** *[subscript] : [subscript] [: stride]*

R6124 stride **is** *scalar-int-expr*

R6125 vector-subscript **is** *int-expr*

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

R6126 allocate-stmt **is** *ALLOCATE (allocation-list* ■
■ *[, STAT = stat-variable])*

R6127 stat-variable **is** *scalar-int-variable*

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

R6128 allocation **is** *allocate-name (explicit-shape-spec-list)*

Constraint: *allocate-name* must be the name of a pointer or an allocatable array.

Constraint: A bound in an *allocation explicit-shape-spec* is not restricted to a specification expression, but must not be an expression involving as a primary an array inquiry function (13.10.15) whose argument is any other object in the same ALLOCATE statement.

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

R6129 nullify-stmt **is** *NULLIFY (pointer-name-list)*

Constraint: Each *pointer-name* must be the name of a pointer.

R6130 deallocate-stmt **is** *DEALLOCATE (allocate-name-list* ■
■ *[, STAT = stat-variable])*

Constraint: Each *allocate-name* must be the name of a pointer or an allocatable array.

Constraint: If the *stat-variable* is a pointer, it must not be deallocated within the same DEALLOCATE statement.

7. EXPRESSIONS AND ASSIGNMENT

R7131 primary **is** *constant*
or *constant-subobject*
or *variable*
or *array-constructor*
or *structure-constructor*
or *function-reference*
or *(expr)*

R7132 constant-subobject **is** *subobject*

R7133 level-1-expr **is** *[defined-unary-op] primary*

R7134 defined-unary-op **is** *. letter [letter]... .*

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

R7135 mult-operand **is** *level-1-expr [power-op mult-operand]*

R7136 add-operand **is** *[add-operand mult-op] mult-operand*

R7137 level-2-expr **is** *[add-op] add-operand*
or *level-2-expr add-op add-operand*

R7138 power-op **is** **

R7139 mult-op **is** *
or /

R7140 add-op **is** +
or −

R7141 level-3-expr **is** *[level-3-expr concat-op] level-2-expr*

R7142 concat-op **is** *//*

R7143 level-4-expr **is** *[level-3-expr rel-op] level-3-expr*

R7144 rel-op **is** *.EQ.*
or *.NE.*
or *.LT.*
or *.LE.*
or *.GT.*
or *.GE.*
or *==*
or *<>*
or *<*
or *<=*
or *>*
or *>=*

R7145 and-operand **is** *[not-op] level-4-expr*

R7146 or-operand **is** *[or-operand and-op] and-operand*

R7147 equiv-operand **is** *[equiv-operand or-op] or-operand*

R7148 level-5-expr **is** *[level-5-expr equiv-op] equiv-operand*

R7149 not-op **is** *.NOT.*

R7150 and-op **is** *.AND.*

R7151 or-op **is** *.OR.*

R7152 equiv-op **is** *.EQV.*
or *.NEQV.*

R7153 expr **is** *[expr defined-binary-op] level-5-expr*

R7154 defined-binary-op **is** *. letter [letter]... .*

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

R7155 logical-expr **is** *expr*

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

R7156 char-expr **is** *expr*

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

R7157 int-expr **is** *expr*

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

R7158 numeric-expr **is** *expr*

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

R7159 constant-expr **is** *expr*

R7160 char-constant-expr **is** *char-expr*

R7161 int-constant-expr **is** *int-expr*

R7162 logical-constant-expr **is** *logical-expr*

R7163 specification-expr **is** *scalar-int-expr*

R7164 assignment-stmt **is** *variable = expr*

R7165 pointer-assignment-stmt **is** *pointer-name => target*

R7166 target **is** *variable*

Constraint: The *pointer-name* must have the POINTER attribute. The target object must have one of the attributes TARGET or POINTER or it must be a subobject of an object with one of these attributes.

Constraint: The *target* must be of the same type, type parameters, and rank as the pointer.

R7167 where-stmt **is** *WHERE (mask-expr) assignment-stmt*

R7168 where-construct **is** *where-construct-stmt*
[*assignment-stmt*]...
[*elsewhere-stmt*
[*assignment-stmt*]...]
end-where-stmt

R7169 where-construct-stmt **is** *WHERE (mask-expr)*

R7170 mask-expr **is** *logical-expr*

R7171 elsewhere-stmt **is** *ELSEWHERE*

R7172 end-where-stmt **is** *END WHERE*

Constraint: In each *assignment-stmt*, the *mask-expr* and the variable being defined must be arrays of the same shape.

8. EXECUTION CONTROL

R8173 block **is** *[execution-part-construct]...*

R8174 if-construct **is** *if-then-stmt*
block
[*else-if-stmt*
block]...
[*else-stmt*
block]
end-if-stmt

R8175 if-then-stmt **is** *[if-construct-name :] IF (scalar-logical-expr) THEN*

R8176 else-if-stmt **is** *ELSE IF (scalar-logical-expr) THEN [if-construct-name]*

R8177 else-stmt **is** *ELSE [if-construct-name]*

R8178 end-if-stmt **is** *END IF [if-construct-name]*

Constraint: If an *if-construct-name* is present, the same name must be specified on both the *if-then-stmt* and the corresponding *end-if-stmt*. If an *if-construct-name* appears on the *if-then-stmt*, it may also optionally appear on any *else-if-stmt* or *else-stmt* belonging to that *if-construct*.

R8179 if-stmt **is** *IF (scalar-logical-expr) action-stmt*

Constraint: The *action-stmt* in the *if-stmt* must not be an *if-stmt*.

R8180 case-construct **is** *select-case-stmt*
[*case-stmt*
block]...
end-select-stmt

R8181 select-case-stmt **is** *[select-construct-name :] SELECT CASE (case-expr)*

R8182 case-stmt **is** *CASE case-selector [select-construct-name]*

R8183 end-select-stmt **is** *END SELECT [select-construct-name]*

Constraint: If a *select-construct-name* is present, the same name must be specified on both the *select-case-stmt* and the corresponding *end-select-stmt*. If a *select-construct-name* appears on the *select-case-stmt*, it may also optionally appear on any *case-stmt* belonging to that *case-construct*.

R8184 case-expr **is** *scalar-int-expr*
or *scalar-char-expr*
or *scalar-logical-expr*

R8185 case-selector **is** *(case-value-range-list)*
or *DEFAULT*

Constraint: Only one DEFAULT *case-selector* may belong to any given *case-construct*.

R8186 case-value-range **is** *case-value*
or *case-value :*
or *: case-value*
or *case-value : case-value*

R8187 case-value **is** *scalar-int-constant-expr*
or *scalar-char-constant-expr*
or *scalar-logical-constant-expr*

Constraint: For a given *case-construct*, each *case-value* must be of the same type as *case-expr*. For character type, length differences are allowed, but the kind type parameters must be the same.

Constraint: A *case-value-range* using a colon must not be used if *case-expr* is of type logical.

Constraint: For a given *case-construct*, the *case-value-ranges* must not overlap; that is, there must be no possible value of the *case-expr* that matches more than one *case-value-range*.

R8188 do-construct **is** *block-do-construct*
or *nonblock-do-construct*

R8189 block-do-construct **is** *do-stmt*
do-block
end-do

R8190 do-stmt **is** *label-do-stmt*
or *nonlabel-do-stmt*

R8191 label-do-stmt **is** *[do-construct-name :] DO label [loop-control]*

R8192 nonlabel-do-stmt **is** *[do-construct-name :] DO [loop-control]*

R8193 loop-control **is** *[,] do-variable = scalar-numeric-expr ,*
scalar-numeric-expr [, scalar-numeric-expr]
or *[,] WHILE (scalar-logical-expr)*

R8194 do-variable **is** *scalar-variable*

Constraint: The *do-variable* must be a scalar integer, default real, or double precision real named variable.

Constraint: Each *scalar-numeric-expr* in *loop-control* must be of type integer, default real, or double precision real.

R8195 do-block **is** *block*

R8196 end-do **is** *end-do-stmt*
or *continue-stmt*

R8197 end-do-stmt **is** *END DO [do-construct-name]*

Constraint: If the *do-stmt* of a *block-do-construct* is identified by a *do-construct-name*, the corresponding *end-do* must be an *end-do-stmt* specifying the same *do-construct-name*. If the *do-stmt* of a *block-do-construct* does not so specify a *do-construct-name*, the corresponding *end-do* must not specify a *do-construct-name*.

Constraint: If the *do-stmt* is a *nonlabel-do-stmt*, the corresponding *end-do* must be an *end-do-stmt*.

Constraint: If the *do-stmt* is a *label-do-stmt*, the corresponding *end-do* must be identified with the same *label*.

R8198 *nonblock-do-construct* **is** *action-term-do-construct*
or *outer-shared-do-construct*

R8199 *action-term-do-construct* **is** *label-do-stmt*
do-body
do-term-action-stmt

R8200 *do-body* **is** [*execution-part-construct*]...

R8201 *do-term-action-stmt* **is** *action-stmt*

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-stmt*, an *arithmetic-if-stmt*, or an *assigned-goto-stmt*.

Constraint: The *do-term-action-stmt* must be identified with a label and the corresponding *label-do-stmt* must refer to the same label.

R8202 *outer-shared-do-construct* **is** *label-do-stmt*
do-body
shared-term-do-construct

R8203 *shared-term-do-construct* **is** *outer-shared-do-construct*
or *inner-shared-do-construct*

R8204 *inner-shared-do-construct* **is** *label-do-stmt*
do-body
do-term-shared-stmt

R8205 *do-term-shared-stmt* **is** *action-stmt*

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 *arithmetic-if-stmt*, or an *assigned-goto-stmt*.

Constraint: The *do-term-shared-stmt* must be identified with a label and all of the *label-do-stmts* of the *shared-term-do-construct* must refer to the same label.

R8206 *cycle-stmt* **is** CYCLE [*do-construct-name*]

Constraint: If a *cycle-stmt* refers to a *do-construct-name*, it must be within the range of that *do-construct*; otherwise, it must be within the range of at least one *do-construct*

R8207 *exit-stmt* **is** EXIT [*do-construct-name*]

Constraint: If an *exit-stmt* refers to a *do-construct-name*, it must be within the range of that *do-construct*; otherwise, it must be within the range of at least one *do-construct*.

R8208 *goto-stmt* **is** GO TO *label*

Constraint: The *label* must be the statement label of a branch target statement that appears in the same scoping unit as the *goto-stmt*.

R8209 *computed-goto-stmt* **is** GO TO (*label-list*) [,] *scalar-int-expr*

Constraint: Each *label* in *label-list* must be the statement label of a branch target statement that appears in the same scoping unit as the *computed-goto-stmt*.

R8210 *assign-stmt* **is** ASSIGN *label* TO *scalar-int-variable*

Constraint: The *label* must be the statement label of a branch target statement or *format-stmt* that appears in the same scoping unit as the *assign-stmt*.

R8211 *assigned-goto-stmt* **is** GO TO *scalar-int-variable* [[,] (*label-list*)]

Constraint: Each *label* in *label-list* must be the statement label of a branch target statement that appears in the same scoping unit as the *assigned-goto-stmt*.

R8212 *arithmetic-if-stmt* **is** IF (*scalar-numeric-expr*) *label*, *label*, *label*

Constraint: Each *label* must be the label of a branch target statement that appears in the same scoping unit as the *arithmetic-if-stmt*.

Constraint: The *scalar-numeric-expr* must not be of type complex.

R8213 *continue-stmt* **is** CONTINUE

R8214 *stop-stmt* **is** STOP [*stop-code*]

R8215 *stop-code* **is** *scalar-char-constant*
or *digit* [*digit* [*digit* [*digit* [*digit*]]]]

R8216 *pause-stmt* **is** PAUSE [stop-code]

9. INPUT/OUTPUT STATEMENTS

R9217 *io-unit* **is** *external-file-unit*
or *
or *internal-file-unit*

R9218 *external-file-unit* **is** *scalar-int-expr*

R9219 *internal-file-unit* **is** *char-variable*

Constraint: The *char-variable* must not be an array section with a vector subscript.

R9220 *open-stmt* **is** OPEN (*connect-spec-list*)

R9221 *connect-spec* **is** [UNIT=] *external-file-unit*
or IOSTAT= *scalar-int-variable*
or ERR= *label*
or FILE= *file-name-expr*
or STATUS= *scalar-char-expr*
or ACCESS= *scalar-char-expr*
or FORM= *scalar-char-expr*
or RECL= *scalar-int-expr*
or BLANK= *scalar-char-expr*
or POSITION= *scalar-char-expr*
or ACTION= *scalar-char-expr*
or DELIM= *scalar-char-expr*
or PAD= *scalar-char-expr*

R9222 *file-name-expr* **is** *scalar-char-expr*

Constraint: If the optional characters UNIT= are omitted from the unit specifier, the unit specifier must be the first item in the *connect-spec-list*.

Constraint: Each specifier must not appear more than once in a given *open-stmt*; an *external-file-unit* must be specified.

Constraint: The *label* used in the ERR= specifier must be the statement label of a branch target statement that appears in the same scoping unit as the OPEN statement.

R9223 *close-stmt* **is** CLOSE (*close-spec-list*)

R9224 *close-spec* **is** [UNIT=] *external-file-unit*
or IOSTAT= *scalar-int-variable*
or ERR= *label*
or STATUS= *scalar-char-expr*

Constraint: If the optional characters UNIT= are omitted from the unit specifier, the unit specifier must be the first item in the *close-spec-list*.

Constraint: Each specifier must not appear more than once in a given *close-stmt*; an *external-file-unit* must be specified.

Constraint: The *label* used in the ERR= specifier must be the statement label of a branch target statement that appears in the same scoping unit as the CLOSE statement.

R9225 *read-stmt* **is** READ (*io-control-spec-list*) [*input-item-list*]
or READ *format* [, *input-item-list*]

R9226 *write-stmt* **is** WRITE (*io-control-spec-list*) [*output-item-list*]

R9227 *print-stmt* **is** PRINT *format* [, *output-item-list*]

R9228 *io-control-spec* **is** [UNIT=] *io-unit*
or [FMT=] *format*
or [NML=] *namelist-group-name*
or REC= *scalar-int-expr*
or IOSTAT= *scalar-int-variable*
or ERR= *label*
or END= *label*
or NULLS= *scalar-int-variable*
or ADVANCE= *scalar-char-expr*
or SIZE= *scalar-int-expr*
or EOR= *label*

Constraint: An *io-control-spec-list* must contain exactly one *io-unit* and may contain at most one of each of the other specifiers.

Constraint: An END= or a NULLS= specifier must not appear in a *write-stmt*.

Constraint: The *label* used in the ERR=, EOR=, or END= specifier must be the statement label of a branch target statement that appears in the same scoping unit as the data transfer statement.

Constraint: A *namelist-group-name* must not be present if an *input-item-list* or an *output-item-list* is present in the data transfer statement.

Constraint: An *io-control-spec-list* must not contain both a *format* and a *namelist-group-name*.

Constraint: If the optional characters UNIT= are omitted from the unit specifier, the unit specifier must be the first item in the control information list.

Constraint: If the optional characters FMT= are omitted from the format specifier, the format specifier must be the second item in the control information list and the first item must be the unit specifier without the optional characters UNIT=.

Constraint: If the optional characters NML= are omitted from the namelist specifier, the namelist specifier must be the second item in the control information list and the first item must be the unit specifier without the optional characters UNIT=.

Constraint: If the unit specifier specifies an internal file, the *io-control-spec-list* must not contain a REC= specifier or a *namelist-group-name*.

Constraint: A NULLS= specifier may be present only in a list-directed input statement.

Constraint: If the REC= specifier is present, an END= specifier must not appear and the *format* must not be an asterisk specifying list-directed input/output.

Constraint: An ADVANCE= and an EOR= specifier may be present only in a formatted sequential READ or WRITE statement that does not contain a namelist specifier or an internal file unit specifier.

Constraint: A SIZE= specifier may be present only in a nonadvancing formatted sequential READ statement that does not contain a namelist specifier or an internal file unit specifier.

R9229 *format* **is** *char-expr*
or *label*
or *
or scalar-int-variable

Constraint: The *label* must be the label of a FORMAT statement that appears in the same scoping unit as the statement containing the format specifier.

R9230 *input-item* **is** *variable*
or *io-implied-do*

R9231 *output-item* **is** *expr*
or *io-implied-do*

R9232 *io-implied-do* **is** (*io-implied-do-object-list* , *io-implied-do-control*)

R9233 *io-implied-do-object* **is** *input-item*
or *output-item*

R9234 *io-implied-do-control* **is** *do-variable* = *scalar-numeric-expr* , ■
▒ *scalar-numeric-expr* [, *scalar-numeric-expr*]

Constraint: The *do-variable* must be a scalar of type integer, default real, or double precision real.

Constraint: In an *input-item-list*, an *io-implied-do-object* must be an *input-item*. In an *output-item-list*, an *io-implied-do-object* must be an *output-item*.

R9235 *backspace-stmt* **is** BACKSPACE *external-file-unit*
or BACKSPACE (*position-spec-list*)

R9236 *endfile-stmt* **is** ENDFILE *external-file-unit*
or ENDFILE (*position-spec-list*)

R9237 *rewind-stmt* **is** REWIND *external-file-unit*
or REWIND (*position-spec-list*)

R9238 *position-spec* **is** [UNIT =] *external-file-unit*
or IOSTAT = *scalar-int-variable*
or ERR = *label*

Constraint: The *label* used in the ERR= specifier must be the statement label of a branch target statement that appears in the same scoping unit as the file positioning statement.

Constraint: If the optional characters UNIT= are omitted from the unit specifier, the unit specifier must be the first item in the *position-spec-list*.

Constraint: A *position-spec-list* must contain exactly one *external-file-unit* and may contain at most one of each of the other specifiers.

R9239 *inquire-stmt* **is** INQUIRE (*inquire-spec-list*)
or INQUIRE (IOLENGTH = *scalar-int-variable*) *output-item-list*

R9240 *inquire-spec* **is** [UNIT =] *external-file-unit*
or FILE = *file-name-expr*
or IOSTAT = *scalar-int-variable*
or ERR = *label*
or EXIST = *scalar-logical-variable*
or OPENED = *scalar-logical-variable*
or NUMBER = *scalar-int-variable*
or NAMED = *scalar-logical-variable*
or NAME = *scalar-char-variable*
or ACCESS = *scalar-char-variable*
or SEQUENTIAL = *scalar-char-variable*
or DIRECT = *scalar-char-variable*
or FORM = *scalar-char-variable*
or FORMATTED = *scalar-char-variable*
or UNFORMATTED = *scalar-char-variable*
or RECL = *scalar-int-variable*
or NEXTREC = *scalar-int-variable*
or BLANK = *scalar-char-variable*

or POSITION = *scalar-char-variable*
or ACTION = *scalar-char-variable*
or READ = *scalar-char-variable*
or WRITE = *scalar-char-variable*
or READWRITE = *scalar-char-variable*
or DELIM = *scalar-char-variable*
or PAD = *scalar-char-variable*

Constraint: An *inquire-spec-list* must contain one FILE= specifier or one UNIT= specifier, but not both, and at most one of each of the other specifiers.

Constraint: In the inquire by unit form of the INQUIRE statement, if the optional characters UNIT= are omitted from the unit specifier, the unit specifier must be the first item in the *inquire-spec-list*.

10. INPUT/OUTPUT EDITING

R10241 *format-stmt* **is** FORMAT *format-specification*

R10242 *format-specification* **is** ([*format-item-list*])

Constraint: The *format-stmt* must be labeled.

Constraint: The comma used to separate *format-items* in a *format-item-list* may optionally be omitted as follows:

(1) Between a P edit descriptor and an immediately following F, E, EN, D, or G edit descriptor (10.6.5)

(2) Before a slash edit descriptor when the optional repeat specification is not present (10.6.2)

(3) After a slash edit descriptor

(4) Before or after a colon edit descriptor (10.6.3)

R10243 *format-item* **is** [*r*] *data-edit-desc*
or *control-edit-desc*
or *char-string-edit-desc*
or [*r*] (*format-item-list*)

R10244 *r* **is** *int-literal-constant*

Constraint: *r* must be positive.

R10245 *data-edit-desc* **is** I *w* [. *m*]
or B *w* [. *m*]
or O *w* [. *m*]
or Z *w* [. *m*]
or F *w* . *d*
or E *w* . *d* [E *e*]
or EN *w* . *d* [E *e*]
or G *w* . *d* [E *e*]
or L *w*
or A [*w*]
or D *w* . *d*
or B *w*
or O *w*
or Z *w*

R10246 *w* **is** *int-literal-constant*

R10247 *m* **is** *int-literal-constant*

R10248 *d* **is** *int-literal-constant*

R10249 *e* **is** *int-literal-constant*

Constraint: *w* and *e* must be positive and *d* and *m* must be zero or positive.

R10250 *control-edit-desc* **is** *position-edit-desc*
or [*r*] /
or :
or *sign-edit-desc*
or *k* P
or *blank-interp-edit-desc*

R10251 *k* **is** *signed-int-literal-constant*

R10252 *position-edit-desc* **is** T *n*
or TL *n*
or TR *n*
or *n* X

R10253 *n* **is** *int-literal-constant*

Constraint: *n* must be positive.

R10254 *sign-edit-desc* **is** S
or SP
or SS

R10255 *blank-interp-edit-desc* **is** BN
or BZ

R10256 *char-string-edit-desc* **is** *char-literal-constant*
or *c* H *rep-char* [*rep-char*]...

R10257 *c* **is** *int-literal-constant*

Constraint: *c* must be positive.

11. PROGRAM UNITS

R11258 *main-program* **is** [*program-stmt*]
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-program-stmt

R11259 *program-stmt* **is** PROGRAM *program-name*

R11260 *end-program-stmt* **is** END [PROGRAM [*program-name*]]

Constraint: In a *main-program*, the *execution-part* must not contain a RETURN statement or an ENTRY statement.

Constraint: The *program-name* may be included in the *end-program-stmt* only if the optional *program-stmt* is used and, if included, must be identical to the *program-name* specified in the *program-stmt*.

R11261 *module* **is** *module-stmt*
[*specification-part*]
[*module-subprogram-part*]
end-module-stmt

R11262 *module-stmt* **is** MODULE *module-name*

R11263 *end-module-stmt* **is** END [MODULE [*module-name*]]

Constraint: If the *module-name* is specified in the *end-module-stmt*, it must be identical to the *module-name* specified in the *module-stmt*.

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

Constraint: An automatic object must not appear in the *specification-part* (R203) of a module.

R11264 *use-stmt* **is** USE *module-name* [, *rename-list*]
or USE *module-name* , ONLY : [*only-list*]

R11265 *rename* **is** *local-name* => *use-name*

R11266 *only* **is** [*local-name*] => *use-name*

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

R11267 *block-data* **is** *block-data-stmt*
[*specification-part*]
end-block-data-stmt

R11268 *block-data-stmt* **is** BLOCK DATA [*block-data-name*]

R11269 *end-block-data-stmt* **is** END [BLOCK DATA [*block-data-name*]]

Constraint: The *block-data-name* may be included in the *end-block-data-stmt* only if it was provided in the *block-data-stmt* and, if included, must be identical to the *block-data-name* in the *block-data-stmt*.

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

12. PROCEDURES

R12270 *interface-block* **is** *interface-stmt*
[*procedure-interface*] ...
[*module-procedure-stmt*] ...
end-interface-stmt

R12271 *interface-stmt* **is** INTERFACE [*generic-spec*]

R12272 *end-interface-stmt* **is** END INTERFACE

R12273 *procedure-interface* **is** *procedure-heading*
[*use-stmt*] ...
[*implicit-part*]
[*declaration-construct*] ...
procedure-ending

R12274 *module-procedure-stmt* **is** MODULE PROCEDURE *procedure-name-list*

R12275 *generic-spec* **is** *generic-name*
or OPERATOR (*defined-operator*)
or ASSIGNMENT (=)

Constraint: An *interface-block* must not contain an *entry-stmt*.

Constraint: The MODULE PROCEDURE specification is allowed only if the *interface-block* has a *generic-spec*.

Constraint: An interface block must not appear in a BLOCK DATA program unit.

R12276 *external-stmt* **is** EXTERNAL *external-name-list*

R12277 *intrinsic-stmt* **is** INTRINSIC *intrinsic-procedure-name-list*

R12278 *function-reference* **is** *function-name* ([*actual-arg-spec-list*])
or *defined-operation*

Constraint: The *actual-arg-spec-list* for a function reference must not contain an *alt-return-spec*.

R12279 *defined-operation* **is** [*actual-arg*] *defined-operator actual-arg*

R12280 *subroutine-reference* **is** *call-stmt*
or *defined-assignment*

R12281 *call-stmt* **is** CALL *subroutine-name* [([*actual-arg-spec-list*])]

R12282 *defined-assignment* **is** *actual-arg* = *actual-arg*

Constraint: *actual-arg* for a *defined-operation* or *defined-assignment* must not be a procedure name or *alt-return-spec*.

R12283 *actual-arg-spec* **is** [*keyword* =] *actual-arg*

R12284 *keyword* **is** *dummy-arg-name*

R12285 *actual-arg* **is** *expr*
or *variable*
or *procedure-name*
or alt-return-spec

R12286 *alt-return-spec* **is** * label

Constraint: The *keyword* may be omitted from an *actual-arg-spec* only if the *keyword* has been omitted from each preceding *actual-arg-spec* in the argument list.

Constraint: Each *keyword* must be the name of a dummy argument in the explicit interface of the procedure.

Constraint: A *procedure-name actual-arg* must not be the name of an internal procedure and must not be the name of an intrinsic subroutine (13.9). If it is the name of an intrinsic function, it must be a specific name for the function (13.12).

Constraint: The *label* used in the *alt-return-spec* must be the statement label of a branch target statement that appears in the same scoping unit as the *call-stmt*.

R12287 *external-subprogram* **is** *procedure-heading*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
procedure-ending

R12288 *procedure-heading* **is** *function-stmt*
or *subroutine-stmt*

R12289 *procedure-ending* **is** *end-function-stmt*
or *end-subroutine-stmt*

R12290 *function-stmt* **is** [*prefix*] FUNCTION *function-name*
([*dummy-arg-name-list*]) [RESULT (*result-name*)]

R12291 *prefix* **is** *type-spec* [RECURSIVE]
or RECURSIVE [*type-spec*]

R12292 *end-function-stmt* **is** END [FUNCTION [*function-name*]]

Constraint: For a function subprogram, the *procedure-heading* must be a *function-stmt* and the *procedure-ending* must be an *end-function-stmt*.

Constraint: FUNCTION must be present on the *end-function-stmt* of an internal or module function.

Constraint: An internal function must not contain an ENTRY statement.

Constraint: If a *function-name* is present on the *end-function-stmt*, it must be identical to the *function-name* specified in the *function-stmt*.

R1217 *external-subprogram* **is** *procedure-heading*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
procedure-ending

R1218 *procedure-heading* **is** *function-stmt*
or *subroutine-stmt*

R1219 *procedure-ending* **is** *end-function-stmt*
or *end-subroutine-stmt*

R12293 *subroutine-stmt* **is** [RECURSIVE] SUBROUTINE *subroutine-name* ■
■ [([*dummy-arg-list*])]

R12294 *dummy-arg* **is** *dummy-arg-name*
or *

R12295 *end-subroutine-stmt* **is** END [SUBROUTINE [*subroutine-name*]]

Constraint: For a subroutine subprogram, the *procedure-heading* must be a *subroutine-stmt* and the *procedure-ending* must be an *end-subroutine-stmt*.

Constraint: SUBROUTINE must be present on the END statement of an internal or module subroutine.

Constraint: An internal subroutine must not contain an ENTRY statement.

Constraint: If a *subroutine-name* is present on the *end-subroutine-stmt*, it must be identical to the *subroutine-name* specified in the *subroutine-stmt*.

R12296 *entry-stmt* **is** ENTRY *entry-name* [([*dummy-arg-list*])]

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

Constraint: A *dummy-arg* may be an alternate return indicator only if the ENTRY statement is contained in a subroutine subprogram.

R12297 *return-stmt* **is** RETURN [*scalar-int-expr*]

Constraint: The *return-stmt* must be contained in the scoping unit of a function or subroutine subprogram.

Constraint: The *scalar-int-expr* is allowed only in the scoping unit of a subroutine subprogram.

R12298 *contains-stmt* **is** CONTAINS

R12299 *stmt-function-stmt* **is** *function-name* ([*dummy-arg-name-list*]) = *scalar-expr*

Constraint: The *scalar-expr* may be composed only of constants (literal and named), references to scalar variables and array elements, references to functions and function dummy procedures, and intrinsic operators. If a reference to another statement function appears in *scalar-expr*, its definition must have been provided earlier in the scoping unit.

Constraint: Named constants in *scalar-expr* must have been declared earlier in the scoping unit. If array elements appear in *scalar-expr*, the parent array must have been declared as an array earlier in the scoping unit. If a scalar variable, array element, function reference, or dummy function reference is typed by the implicit typing rules, its appearance in any subsequent type declaration statement must confirm this implied type and the values of any implied type parameters.

Constraint: The *function-name* and each *dummy-arg-name* must be specified, explicitly or implicitly, to be scalar data objects.

Constraint: A given *dummy-arg-name* may appear only once in any *dummy-arg-name-list*.

Constraint: Each scalar variable reference may be either a reference to a dummy argument of the statement function or a reference to a variable within the same scoping unit as the statement function statement.

13. INTRINSIC PROCEDURES

14. SCOPE, ASSOCIATION, AND DEFINITION

APPENDIX E. PERMUTED INDEX FOR HEADINGS

(This appendix is not part of American National Standard X3.9-198x, 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
5.1.2.9. ALLOCATABLE Attribute
6.3.1. ALLOCATE Statement
/Arguments Associated with Alternate Return Indicators
6.3.4. Summary of Array Name Appearances
14.6.1.1. Argument Association
13.3. Positional Arguments or Argument Keywords
14.1.2.5. Argument Keywords
12.4.1. Actual Argument List
Function 13.10.1. Argument Presence Inquiry
Functions 13.4. Argument Presence Inquiry
Characteristics of Dummy Arguments 12.2.1.
Characteristics of Asterisk Dummy Arguments 12.2.1.3.
on Entities Associated with Dummy Arguments /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
2.4.7. 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
Interpretation of Masked Array Assignments 7.5.3.2.
6.2.1.1. Array Constants and Variables
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
6.3.4. Summary of Array Name Appearances
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
Statement 8.2.4. ASSIGN and Assigned GO TO
8.2.4. ASSIGN and Assigned GO TO Statement

Derived-Type Operations and Assignment 4.4.5.
7. EXPRESSIONS AND ASSIGNMENT
7.5. 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
7.5.2. Pointer Assignment Statement
Interpretation of Defined Assignment Statements 7.5.1.6.
14.5. Scope of the Assignment Symbol
7.5.3. Masked Array Assignment WHERE
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
/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
11.2.2. Host Association
12.4.1.4. Sequence Association
14.6. Association
14.6.1. Name Association
14.6.1.1. Argument Association
Use Association and Host Association 14.6.1.2.
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
5.5. Storage Association of Data Objects
Objects 14.6.3.3. Association of Scalar Data
14.6.3.2. Association of Storage Sequences
Functions 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
7.3.2. Binary Defined Operation
13.9.3. Bit Copy Subroutine
13.10.9. Bit Inquiry Functions
Procedures 13.5.7. Bit Manipulation and Inquiry
13.10.10. Bit Manipulation Functions
Mathematical, Character, and Bit Procedures 13.5. Numeric,
between Named Common and Blank Common /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
2.2.4.4. 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
Executable Constructs in Blocks 8.1.1.1.
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
5.1.1.5. CHARACTER
13.5. Numeric, Mathematical, Character, and Bit Procedures
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
3.1.5. Character Graphics
13.5.4. Character Inquiry Function
13.10.5. Character Inquiry Functions
7.1.7.4. Evaluation of the Character Intrinsic Operation
7.2.2. Character Intrinsic Operation
5.5.1.2. Equivalence of Character Objects
3.1. Fortran 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
3.1.6. Representable 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
STATUS= Specifier in the CLOSE Statement 9.3.5.1.
3.1.7. 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
/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
5.1.1.4. COMPLEX
10.5.1.2. Real and Complex Editing
10.5.1.2.4. Complex Editing
Generalized Real and Complex Editing 10.5.4.1.2.
7.2.1.2. Complex Exponentiation
4.3.1.3. Complex Type
14.1.2.4. Components
6.1.2. Structure Components
8.2.3. Computed GO TO Statement
4.1. The Concept of Type
2. FORTRAN TERMS AND CONCEPTS
2.2. Program Unit Concepts
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
2.4.4. Constant
10.7.1. Character Constant Edit Descriptor
7.1.6.1. Constant Expression
3.2.3. Constants
4.1.2. Constants
6.2.1.1. Array Constants and Variables

8.1.2. IF Construct
8.1.2.1. Form of the IF Construct
8.1.2.2. Execution of an IF Construct
8.1.3. CASE Construct
8.1.3.1. Form of the CASE Construct
8.1.3.2. Execution of a CASE Construct
8.1.4. DO Construct
8.1.4.1. Forms of the DO Construct
8.1.4.1.1. Form of the Block DO Construct
Form of the Nonblock DO Construct 8.1.4.1.2.
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
9.4.1.10. Nulls Count
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
Characteristics of Dummy Data Objects 12.2.1.1.
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
5.2.6. DATA Statement
11.3.3.3. Data Structures
File Position Prior to Data Transfer 9.2.1.3.2.
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
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
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
2.5.3. Declaration
5.1. Type Declaration Statements

5. DATA OBJECT DECLARATIONS AND SPECIFICATIONS
5.1.2.4.3. Deferred-Shape Array
7.5.1.3. Defined Assignment Statement
7.5.1.6. Interpretation of Defined Assignment Statements
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
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
Other Than Fortran 12.5.3. Definition of Procedures by Means
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
4.3.1.2. Real and Double Precision Real Type
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
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. 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
12.4.5. Elemental Subroutine Reference
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
Error, End-of-Record, and End-of-File Conditions 9.4.3.
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
Types and Values to Objects and Entities 4.2. Relationship of
12.5.2.9. Restrictions on Entities Associated with Dummy/
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
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
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
1.3.2. Exclusions
Blocks 8.1. Executable Constructs Containing
8.1.1.1. Executable Constructs in Blocks
11.1.2. Main Program Executable Part
2.2.2. 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

2.2.4.1. External Procedure
12.3.2.2. EXTERNAL Statement
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
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
9.4.4.4.2. 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
3.1. Fortran Character Set
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

Argument Presence Inquiry Function 13.10.1.
13.10.11. Transfer Function
13.10.17. Array Reshape Function
13.10.6. Kind Inquiry Function
13.5.4. Character Inquiry 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
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
Association Status Inquiry Functions 13.10.20. Pointer
13.10.3. Mathematical Functions
13.10.4. Character Functions
13.10.5. Character Inquiry Functions
13.10.7. Logical Functions
13.10.8. Numeric Inquiry Functions
13.10.9. Bit Inquiry Functions
Specific Names for Intrinsic Functions 13.12.
13.4. Argument Presence Inquiry Functions
13.5.1. Numeric Functions
13.5.2. Mathematical Functions
13.5.3. Character Functions
13.5.5. Kind Inquiry Functions
13.5.6. Logical 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
13.10. Generic Intrinsic Functions
11.3.3.4. Global Allocatable Arrays
11.3.3.2. Global Data
14.1.1. Global Entities
8.1.1. Rules Governing Blocks
3.1.5. Character Graphics
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
11.2.2. Host Association
14.6.1.2. Use Association and Host Association
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

Associated with Alternate Return Indicators 12.4.1.3. Arguments
9.4.1. Control Information List
8.1.4.4.1. Loop Initiation
10.8.1. List-Directed Input
10.9.1. Namelist Input
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
9.6.1.1. FILE= Specifier in the INQUIRE Statement
FORM= Specifier in the INQUIRE Statement 9.6.1.10.
FORMATTED= Specifier in the INQUIRE Statement 9.6.1.11.
UNFORMATTED= Specifier in the INQUIRE Statement 9.6.1.12.
RECL= Specifier in the INQUIRE Statement 9.6.1.13.
NEXTREC= Specifier in the INQUIRE Statement 9.6.1.14.
BLANK= Specifier in the INQUIRE Statement 9.6.1.15.
POSITION= Specifier in the INQUIRE Statement 9.6.1.16.
ACTION= Specifier in the INQUIRE Statement 9.6.1.17.
READ= Specifier in the INQUIRE Statement 9.6.1.18.
WRITE= Specifier in the INQUIRE Statement 9.6.1.19.
EXIST= Specifier in the INQUIRE Statement 9.6.1.2.
READWRITE= Specifier in the INQUIRE Statement 9.6.1.20.
DELIM= Specifier in the INQUIRE Statement 9.6.1.21.
9.6.1.22. PAD= Specifier in the INQUIRE Statement
OPENED= Specifier in the INQUIRE Statement 9.6.1.3.
NUMBER= Specifier in the INQUIRE Statement 9.6.1.4.
NAMED= Specifier in the INQUIRE Statement 9.6.1.5.
9.6.1.6. NAME= Specifier in the INQUIRE Statement
ACCESS= Specifier in the INQUIRE Statement 9.6.1.7.
SEQUENTIAL= Specifier in the INQUIRE Statement 9.6.1.8.
DIRECT= Specifier in the INQUIRE Statement 9.6.1.9.
IOLENGTH= Specifier in the INQUIRE Statement 9.6.3.
9.6. File Inquiry
13.10.1. Argument Presence Inquiry Function
13.10.6. Kind Inquiry Function
13.5.4. Character Inquiry Function
13.10.15. Array Inquiry Functions
Pointer Association Status Inquiry Functions 13.10.20.
13.10.5. Character Inquiry Functions
13.10.8. Numeric Inquiry Functions
13.10.9. Bit Inquiry Functions
13.4. Argument Presence Inquiry Functions
13.5.5. Kind 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
2.2.4.4. 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
2.2.4.3. Internal Procedure
11.1.3. Main Program Internal Procedures

11.2.1. 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
5.1.2.11. 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
Evaluation of Relational Intrinsic Operations 7.1.7.5.
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
13.11. Intrinsic Subroutines
13.9. Intrinsic Subroutines
2.4.1.1. Intrinsic Type
1. INTRODUCTION
INQUIRE Statement 9.6.3. IOLENGTH= Specifier in the
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.
14.1.2.5. Argument Keywords
3.2.1. Keywords
13.10.6. Kind Inquiry Function
13.5.5. Kind Inquiry Functions
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
Data Transfer Input/Output List 9.4.2.
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
10.8.2. 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
10.5.2. Logical Editing
10.5.4.2. Generalized Logical Editing
13.10.7. Logical Functions

13.5.6. Logical Functions
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
2.2.3. 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
Procedures 13.5. Numeric, Mathematical, Character, and Bit
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
Explicit Format Specification Methods 10.1.
13.7.1. Models for Integer and Real Data
2.2.5. Module
2.2.4.2. Module Procedure
External, Internal, and Module Procedures 12.1.2.2.
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
6.3.4. Summary of Array Name Appearances
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
10.9.1.2. 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
12.5.5. Overloading Names
14.1. Scope of Names
3.2.2. Names
Designators 5.5.1.3. Array Names and Array Element
13.12. Specific Names for Intrinsic Functions
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
1.5. Notation Used in This Standard
10.8.1.1. Null Values
10.9.1.4. Null Values
6.3.2. NULLIFY Statement
9.4.1.10. Nulls Count
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
10.5.4.1. 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
and Bit Procedures 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.
Storage Association of Data Objects 5.5.
Equivalence of Character Objects 5.5.1.2.
6. USE OF DATA OBJECTS
/of Types and Values to Objects and Entities
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
STATUS= Specifier in the OPEN Statement 9.3.4.2.
ACCESS= Specifier in the OPEN Statement 9.3.4.3.
9.3.4.4. FORM= Specifier in the OPEN Statement
9.3.4.5. RECL= Specifier in the OPEN Statement
BLANK= Specifier in the OPEN Statement 9.3.4.6.
POSITION= Specifier in the OPEN Statement 9.3.4.7.
ACTION= Specifier in the OPEN Statement 9.3.4.8.
DELIM= Specifier in the OPEN Statement 9.3.4.9.
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
Evaluation of a Defined Operation 7.1.7.7.
7.2.2. Character Intrinsic Operation
7.3.1. Unary Defined Operation
7.3.2. Binary Defined Operation
4.1.3. Operations
7.1.2. Intrinsic Operations
7.1.3. Defined Operations
Rules for Intrinsic Operations /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.
Interpretation of Intrinsic Operations 7.2.
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
10.9.2.2. Namelist Output Records
14.1.2.3. Procedure Overloading
12.5.5. Overloading Names
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
5.2.7. PARAMETER Statement
7.1.4.1. Data Type, Type Parameters, and Shape of a/
7.1.4. Data Type, Type Parameters, and Shape of an/
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
2.4.8. Pointer
7.5.2. Pointer Assignment Statement
14.6.2. Pointer Association
Inquiry Functions 13.10.20. Pointer Association Status
Inquiry Functions 13.8.10. Pointer Association Status
5.1.2.7. POINTER Attribute
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
4.3.1.2. Real and Double Precision Real Type
9.3.3. Preconnection
13.10.1. Argument Presence Inquiry Function
13.4. Argument Presence Inquiry Functions
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
2.2.4. Procedure
2.2.4.1. External Procedure
2.2.4.2. Module Procedure
2.2.4.3. Internal Procedure
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
2.2.4.4. Procedure Interface Block
11.3.3.5. Procedure Libraries
14.1.2.3. Procedure Overloading
12.4. Procedure Reference
11.1.3. Main Program Internal Procedures
11.2. Procedures
11.2.1. Internal Procedures
12. PROCEDURES
12.1.2.1. Intrinsic Procedures
External, Internal, and Module Procedures 12.1.2.2.
12.1.2.3. Dummy Procedures
12.2. Characteristics of Procedures
Characteristics of Dummy Procedures 12.2.1.2.
Arguments Associated with Dummy Procedures 12.4.1.2.
13. INTRINSIC PROCEDURES
Specifications of the Intrinsic Procedures 13.13.
13.2. Elemental Intrinsic Procedures
Mathematical, Character, and Bit Procedures 13.5. Numeric,
Bit Manipulation and Inquiry Procedures 13.5.7.
Fortran 12.5.3. Definition of Procedures by Means Other Than
12.5.2. Procedures Defined by Subprograms
1.2. Processor
9.2.2.1. Internal File Properties
13.9.2. Pseudorandom Numbers
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
Type 4.3.1.2. Real and Double Precision Real
13.7.1. Models for Integer and Real Data
Real and Double Precision Real Type 4.3.1.2.
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
12.4.5. Elemental Subroutine Reference
2.5.5. Reference
9.7. Restrictions on Function References and List Items
7.1.7.5. Evaluation of Relational Intrinsic Operations
7.2.3. Relational Intrinsic Operations
to Objects and Entities 4.2. Relationship of Types and Values
3.1.6. Representable Characters
13.10.17. Array Reshape Function
13.8.7. Array Reshape Function
Statements 9.8. Restriction on Input/Output
9.2.2.2. Internal File Restrictions
Equivalence 5.5.2.5. Restrictions on Common and
Not Present 12.5.2.8. Restrictions on Dummy Arguments
Associated with Dummy/ 12.5.2.9. Restrictions on Entities
Statements 5.5.1.4. Restrictions on EQUIVALENCE
References and List Items 9.7. Restrictions on Function
Specifiers 9.6.2. Restrictions on Inquiry
Type Parameters, and Shape of the Result of an Operation /Type,
Characteristics of Function Results 12.2.2.
Intrinsic Function Arguments and Results 13.2.1. Elemental
14.1.2.2. Function 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 7.5.1.4.
7.1.5. Conformability 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
2.4.6. 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
1.3. Scope
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
14.5. Scope of the Assignment Symbol
2.2.1. Scoping Unit
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
3.1.7. Collating Sequence
5.5.2.1. Common Block Storage Sequence
12.4.1.4. Sequence Association
Association of Storage Sequences 14.6.3.2.
9.2.1.2.1. Sequential Access
INQUIRE Statement 9.6.1.8. SEQUENTIAL= Specifier in the
3.1. Fortran Character Set
4.1.1. Set of Values
Data Type, Type Parameters, and Shape of a Primary 7.1.4.1.
Data Type, Type Parameters, and Shape of an Expression 7.1.4.
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
CHARACTERS, LEXICAL TOKENS, AND SOURCE FORM 3.
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

Functions 13.12. Specific Names for Intrinsic
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. Specification of the Procedure
5.2. Attribute Specification Statements
11.1.1. Main Program Specifications
5. DATA OBJECT DECLARATIONS AND SPECIFICATIONS
Procedures 13.13. 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
Statement 9.6.1.10. FORM= Specifier in the INQUIRE
Statement 9.6.1.11. FORMATTED= Specifier in the INQUIRE
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
Statement 9.6.3. IOLENGTH= 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
1.5. 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
Restriction on Input/Output Statements 9.8.
9.4.1.4. Input/Output Status
13.10.20. Pointer Association Status Inquiry Functions
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
2.4.9. 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
2.4.3.2. Subobjects

12.5.2.2. Function Subprogram
12.5.2.3. Subroutine Subprogram
12.5.2.4. Instances of a Subprogram
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 Subroutine Reference
12.5.2.3. Subroutine Subprogram
13.11. Intrinsic Subroutines
13.9. Intrinsic Subroutines
13.9.1. Date and Time Subroutines
6.2.2.5. Vector Subscript
6.2.2.4. Subscript Triplet
6.1.1. Substrings
6.3.4. Summary of Array Name Appearances
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
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
6.2.2.4. Subscript Triplet
2.4.1. Data Type
2.4.1.1. Intrinsic Type
2.4.1.2. Derived Type
4.1. The Concept of Type
4.3.1.1. Integer Type
Real and Double Precision Real Type 4.3.1.2.
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/ 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/ 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
Entities 4.2. Relationship of Types and Values to Objects and
7.3.1. Unary Defined Operation
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
2.2.1. Scoping Unit
Connection of a File to a Unit 9.3.2.
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
Scope of External Input/Output Units 14.3.
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
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 and Entities
2.4.5. Variable
6.2.1.1. Array Constants and Variables
Functions 13.8.3. Vector and Matrix Multiplication
Functions 13.10.13. Vector and Matrix Multiply
6.2.2.5. 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

APPENDIX F. GLOSSARY OF TECHNICAL TERMS

(This appendix is not part of American National Standard X3.9-198x, but is included for information only.)

The following is a list of the principal technical terms used in the standard and their definitions. A reference in parentheses immediately after a term is to the section where the term is defined or explained. The wording of a definition here is not necessarily the same as in the standard. Where the definition uses a term that is itself defined in this glossary, the first occurrence of the term is printed in italics.

action statement. A single *statement* specifying a computational action (R216).

actual argument (12.4.1). An *expression*, a *variable*, a *procedure*, or an alternate return specifier that is specified in a procedure *reference*.

allocatable array (5.1.2.4.3). A *named array* whose *type*, *type parameters*, and *rank* are specified in a *type declaration statement* containing an ALLOCATABLE *attribute*. Only when it has space allocated for it does it have a *shape* and may it be *referenced* or *defined*.

argument (12). An *actual argument* or a *dummy argument*.

argument association (14.6.1.1). The relationship between an *actual argument* and a *dummy argument* during the execution of a *procedure reference*.

argument keyword (2.5.2). A *dummy argument name*. It may be used in a *procedure reference* ahead of the equals symbol (R1214) provided the procedure has an *explicit interface*.

array (2.4.7). A set of *data*, all of the same *type* and *type parameters*, whose individual elements are arranged in a rectangular pattern. It may be a *named* array, the target of an array pointer, an *array section*, a *structure component*, a *function* value, or an *expression*. Its *rank* is at least one.

array element (2.4.7, 6.2.2.1). One of the *scalar data* that make up an *array*.

array pointer (5.1.2.2.3). A *pointer* to an *array*.

array section (6.2.2.3). A *subobject* of an *array* consisting of regularly spaced *array elements* or *substrings* of regularly spaced array elements.

array-valued. Having the property of being an *array*.

assignment statement (7.5.1.1). A *statement* of the form '*variable* = *expression*'.

association (14.6). *Name association*, *pointer association*, or *storage association*.

assumed-size array (5.1.2.4.4). A *dummy array* whose *size* is assumed from the associated *actual argument*. Its last upper bound is specified by an asterisk.

attribute (5). A property of a *data object* that may be specified in a *type declaration statement* (R501).

belong (8.1.4.4.3, 8.1.4.4.4). If an EXIT or a CYCLE *statement* contains a *construct name*, the statement **belongs** to the DO construct using that name. Otherwise, it **belongs** to the innermost DO construct in which it appears.

block (8.1). A sequence of *executable constructs* embedded in another executable construct, bounded by *statements* that are particular to the construct, and treated as an integral unit.

block data program unit (11.4). A *program unit* that provides initial values for *data objects* in *named common blocks*.

bounds (5.1.2.4.1). For a *named array*, the limits within which the values of the *subscripts* of its *array elements* must lie.

character. A letter, digit, or other symbol.

characteristics (12.2)

(1) Of a *procedure*, its classification as a *function* or *subroutine*, the characteristics of its *dummy arguments*, and the characteristics of its *function result* if it is a function.

(2) Of a *dummy argument*, whether it is a *data object*, a *procedure*, or an alternate return indicator; and whether it has the OPTIONAL *attribute*.

character string (4.3.2.1). A sequence of *characters* numbered from left to right 1, 2, 3, . . .

character storage unit (14.6.3.1). The unit of storage for holding a *datum* of *type* character.

common block (5.5.2). A block of physical storage that may be accessed by any of the *scoping units* in an *executable program*.

component (4.4). A constituent of a *derived type*.

conformable (2.4.7). Two *arrays* are said to be **conformable** if they have the same *shape*. A *scalar* is **conformable** with any array.

conformance (1.4). An *executable program* conforms to the standard if it uses only those forms and relationships described therein and if the executable program has an interpretation according to the standard. A *program unit* conforms to the standard if it can be included in an executable program in a manner that allows the executable program to be standard conforming. A *processor* conforms to the standard if it executes standard-conforming programs in a manner that fulfills the interpretations prescribed in the standard.

connected (9.3.2).

(1) For an *external unit*, the property of referring to an *external file*.

(2) For an *external file*, the property of having an *external unit* that refers to it.

constant (2.4.4). A *data object* whose value must not change during execution of an *executable program*. It may be a *named constant* or a *literal constant*.

constant expression (7.1.6.1). An *expression* satisfying rules that ensure that is value does not vary during program execution.

construct (8). A sequence of *statements* starting with a CASE, DO, IF, or WHERE statement and ending with the corresponding terminal statement.

data. Plural of *datum*.

data entity (2.4.3, 4.2). An *entity* that has or may have a data value. It may be a *constant*, a *variable*, an *expression*, or a *function result*.

data object (2.4.3.1). A *datum* or a set of *data* of the same *type* and *type parameters* that may be *referenced* as a whole. It may be *named*, it may be a *subobject*, or it may be a *literal constant*.

data type (2.4.1). A *named* category of *data* that is characterized by a set of values, together with a way to denote these values and a collection of *operations* that interpret and manipulate the values. A data type may be parameterized, in which case the set of data values depends on the values of the parameters.

datum. A single quantity that may have any of the set of values specified for its *data type*.

definable (2.5.4). A *variable* is **definable** if its value may be changed by the appearance of its *name* or *designator* on the left of an *assignment statement*. An *allocatable array* that has not been allocated is an example of a *data object* that is not definable. An example of a *subobject* that is not definable is C(I) when C is an *array* that is a *constant* and I is an integer variable.

defined (2.5.4). For a *data object*, the property of having or being given a valid value.

defined assignment statement (7.5.1.3). An *assignment statement* that is not an *intrinsic* assignment statement and is defined by a *subroutine subprogram* that has an *explicit interface*.

defined operation (7.1.3). An *operation* that is not an *intrinsic* operation and is defined by a *function subprogram* that has an *explicit interface*.

deleted feature (1.6). A feature in FORTRAN 77 that is considered to have been redundant and largely unused. No features in FORTRAN 77 have been deleted from the standard. Note that a feature designated as an *obsolescent feature* in the standard may become a deleted feature in

(or may not)

the next revision.

derived type (2.4.1.2, 4.4). A *type* whose *data* have *components*, each of which is either of intrinsic type or of another derived type.

designator. See *subobject designator*.

disassociated (2.4.8). A *pointer* is **disassociated** if it is not *pointer associated* with a *target*.

dummy argument (12.5.2.2, 12.5.2.3, 12.5.2.5, 12.5.4). An entity whose *name* appears in the parenthesized list following the *procedure* name in a FUNCTION *statement*, a SUBROUTINE *statement*, an ENTRY statement, or a *statement function* statement.

dummy pointer. A *dummy argument* that is a *pointer*.

dummy procedure (12.1.2.3). A *dummy argument* that is specified or *referenced* as a *procedure*.

dummy array A *dummy argument* that is an *array*.

elemental (12.4.3). An adjective applied to an *operation*, *function*, or *assignment statement* that is applied independently to elements of an *array* or corresponding elements of a set of *conformable* arrays and *scalars*.

entity. The term used for any of the following: a *program unit*, a *procedure*, an *operator*, an *interface block*, a *common block*, an *external unit*, a *statement function*, a *type*, a *named variable*, an *expression*, a *component* of a *structure*, a *named constant*, a *statement label*, a *construct*, or a namelist group.

executable construct (2.1). A CASE, DO, IF, or WHERE *construct* or an *action statement* (R216).

executable program (2.2.2). A set of *program units* that includes exactly one *main program*.

executable statement (2.3.1). An instruction to perform or control one or more computational actions.

explicit interface (12.3.1). For a *procedure referenced* in a *scoping unit*, the property of being an *internal procedure*, a *module procedure*, an *intrinsic procedure*, an *external procedure* that has an *interface block*, or a *dummy procedure* that has an interface block.

explicit-shape array (5.1.2.4.1). A *named array* that is declared with *explicit bounds*.

expression (7.1). A sequence of *operands*, *operators*, and parentheses (R722). It may be a *variable*, a *constant*, a *function reference*, or may represent a computation.

extent (2.4.7). The size of one dimension of an *array*.

external file (9.2.1). A sequence of *records* that exists in a medium external to the *executable program*.

external procedure (2.2.4.1). A *procedure* that is defined by an *external subprogram* or by a means other than Fortran.

external subprogram (2.2). A *subprogram* that is not contained in a *main program*, *module*, or another subprogram.

external unit (9.3). A mechanism that is used to refer to an *external file*. It is identified by a nonnegative integer.

file (9.2). An *internal file* or an *external file*.

function (2.2.4). A *procedure* that is invoked in an *expression*.

function result (12.5.2.2). The *data object* that returns the value of a *function*, beginning with

function subprogram (12.5.2.2). A sequence of *statements* ~~from~~ a FUNCTION statement that is not in an *interface block* to the corresponding END statement.

Part extending

global entity (14.1.1). An *entity* identified by a *lexical token* whose *scope* is an *executable program*. It may be a *program unit*, a *common block*, or an *external procedure*.

host (2.2.4.3). A *main program* or *subprogram* that contains an *internal procedure* is called the **host** of the internal procedure. A *module* that contains a *module procedure* is called the **host** of the module procedure.

host association (11.2.2). The process by which an *internal subprogram*, *module subprogram*, or *derived type* definition accesses *entities* of its *host*.

implicit interface (12.3.1). A *procedure referenced* in a *scoping unit* is said to have an **implicit interface** if the procedure is an *external procedure* that does not have an *interface block*, a *dummy procedure* that does not have an interface block, or a *statement function*.

instance of a subprogram (12.5.2.4). The copy of a *subprogram* that is created when a *procedure* defined by the subprogram is *invoked*.

interface block (12.3.2.1). A sequence of *statements* from an INTERFACE statement to the corresponding END INTERFACE statement.

interface of a procedure (12.3). See *procedure interface*.

internal file (9.2.2). A character *variable* that is used to transfer and convert *data* from internal storage to internal storage.

internal procedure (2.2.4.3). A *procedure* that is defined by an *internal subprogram*.

internal subprogram (2.2). A *subprogram* contained in a *main program* or another subprogram.

intrinsic (2.5.7). An adjective applied to *types*, *operations*, *assignment statements*, and *procedures* that are defined in the standard and may be used in any *scoping unit* without further definition or specification.

invoke (2.2.4).

(1) To call a *subroutine* by a CALL *statement* or by a *defined assignment statement*.

(2) To call a *function* by a *reference* to it by *name* or *operator* during the evaluation of an *expression*.

keyword (2.5.2). *Statement keyword* or *argument keyword*.

label. See *statement label*.

length of a character string (4.3.2.1). The number of *characters* in the *character string*.

lexical token (3.2). A sequence of one or more characters with an indivisible interpretation.

literal constant (2.4.4). A *constant* without a *name*.

local entity (14.1.2). An *entity* identified by a *lexical token* whose *scope* is a *scoping unit*.

main program (2.2.3, 11.1). A *program unit* that is not a *module*, *subprogram*, or *block data program unit*.

many-one array section (6.2.2.5). An *array section* with a *vector subscript* having two or more elements with the same value.

module (2.2.5, 11.3). A *program unit* that contains or accesses definitions to be accessed by other program units.

module procedure (2.2.4.2). A *procedure* that is defined by a *module subprogram*.

module subprogram (2.2). A *subprogram* that is contained in a *module* but is not an *internal subprogram*.

name (3.2.2). A *lexical token* consisting of a letter followed by up to 30 alphanumeric characters (letters, digits, and underscores).

name association (14.6.1). *Argument association*, *use association*, or *host association*.

named. Having a *name*.

named constant (2.4.4). A *constant* that has a *name*.

numeric storage unit (2.4.9). The unit of storage for holding a *datum* of *type* default real, integer, or logical.

object (2.4.3.1). *Data object*.

obsolescent feature (1.6). A feature in FORTRAN 77 that is considered to have been redundant but that is still in frequent use.

operand (2.5.8). An *expression* that precedes or succeeds an *operator*.

operation (7.1.2). A computation involving one or two *operands*.

operator (2.5.8). A *lexical token* that specifies an *operation*.

pointer (2.4.8). 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*.

pointer assignment (7.5.2). The *pointer association* of a *pointer* with a *target* by the execution of a *pointer assignment statement* or the execution of an *assignment statement* for a *data object* of *derived type* having the pointer as a *subobject*.

pointer assignment statement (7.5.2). A *statement* of the form '*pointer-name => target*'.

pointer associated (6.3, 7.5.2). The relationship between a *pointer* an a *target* following a *pointer assignment* or a valid execution of an ALLOCATE *statement*.

pointer association (14.6.2). The process by which a *pointer* becomes *pointer associated* with a *target*.

present (12.5.2.8). A *dummy argument* is **present** in an *instance of a subprogram* if it is *associated* with an *actual argument* and the actual argument is a dummy argument that is present in the invoking *procedure* or is not a dummy argument of the invoking procedure.

procedure (2.2.4, 12.1). A computation that may be *invoked* during program execution. It may be a *function* or a *subroutine*. It may be an *external procedure*, a *module procedure*, an *internal procedure*, a *dummy procedure*, or a *statement function*. A *subprogram* may define more than one procedure if it contains ENTRY *statements*.

procedure interface (12.3). The *characteristics* of a *procedure*, the *name* of the procedure, the name of each *dummy argument*, the *operator* (if any) by which it may be *referenced* (*functions* only), and whether it may be referenced by an *assignment statement* (*subroutines* only).

processor (1.2). The combination of a computing system and the mechanism by which *executable programs* are transformed for use on that computing system.

program. See *executable program* and *main program*.

program unit (2.2). The fundamental component of an *executable program*. A sequence of *statements* and comment lines. It may be a *main program*, a *module*, an *external subprogram*, or a *block data program unit*.

rank (2.4.7). The number of dimensions of an *array*. Zero for a *scalar*.

record (9.1). A sequence of values that is treated as a whole within a *file*.

reference (2.5.5). The appearance of a *data object name* or *subobject designator* in a context requiring the value at that point during execution, or the appearance of a *procedure* name, its *operator* symbol, or a *defined assignment statement* in a context requiring execution of the procedure at that point. Note that neither the act of defining a *variable* nor the appearance of the name of a procedure as an *actual argument* is regarded as a reference.

scalar (2.4.6).

(1) A single *datum* that is not an *array*.

(2) Not having the property of being an *array*.

scope (14). That part of an *executable program* within which a *lexical token* has a single interpretation. It may be an *executable program*, a *scoping unit*, a single *statement*, or a part of a statement.

scoping unit (2.2.1). One of the following:

(1) A *derived type* definition,

(2) An *interface block*, excluding any interface blocks contained within it, or

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

section subscript (6.2.6). A *subscript* or *subscript triplet* in an *array section selector*.

selector. A syntactic mechanism for designating

(1) Part of a *data object*. It may designate a *substring*, an *array element*, an *array section*, or a *structure component*.

(2) The set of values for which a CASE *block* is executed.

shape (2.4.7). For an *array*, the *rank* and *extents*. The shape may be represented by the rank-one array whose elements are the extents in each dimension.

size (2.4.7). For an *array*, the total number of elements.

standard module (1.7). A *module* standardized as a separate collateral standard.

statement (3.3). A sequence of *lexical tokens*. It usually consists of a single line, but the ampersand symbol may be used to continue a statement from one line to another and the semicolon symbol may be used to separate statements within a line.

statement entity (14). An *entity* identified by a *lexical token* whose *scope* is a single *statement* or part of a statement.

statement function (12.5.4). A *procedure* specified by a single *statement* that is similar in form to an *assignment statement*.

statement keyword (2.5.2). A word that is part of the syntax of a *statement* and that may be used to identify the statement.

statement label (3.2.5). A *lexical token* consisting of up to five digits that precedes a *statement* and may be used to refer to the statement.

storage association (14.6.3). The relationship between two *storage sequences* if a storage unit of one is the same as a storage unit of the other.

storage sequence (14.6.3.1). A sequence of contiguous *storage units*.

storage unit (14.6.3.1, 2.4.9). A *character storage unit* or a *numeric storage unit*.

stride (6.2.2.4). The increment specified in a *subscript triplet*.

structure (4.4). A *scalar data object* of *derived type*.

structure component (6.1.2). The part of a *structure* corresponding to a *component* of its *type*.

subobject (2.4.3.2). A portion of a *named data object* that may be *referenced* or *defined* independently of other portions. It may be an *array element*, an *array section*, a *structure component*, or a *substring*.

subobject designator (2.5.1). A *name*, followed by one of more of the following: *component selectors*, *array section* selectors, *array element selectors*, *and substring selectors*.

subprogram (2.2). A *function subprogram* or a *subroutine subprogram*. *Modules* and *block data program units* are not subprograms.

subroutine (2.2.4). A *procedure* that is *invoked* by a CALL *statement* or by a *defined assignment statement*.

subroutine subprogram (12.5.2.3). A sequence of *statements* beginning with a SUBROUTINE statement that is not in an *interface block* to the corresponding END statement.

subscript (6.2.2). One of the list of *scalar* integer *expressions* in an *array element selector*. Note that in FORTRAN 77, the whole list was called the subscript.

subscript triplet (6.2.2). An item in the list of an *array section selector* that contains a colon and specifies a regular sequence of integer values.

substring (6.1.1). A contiguous portion of a *scalar character string*. Note that an *array section* can include a *substring selector*; the result is called an array section and not a substring.

target (5.1.2.8). A *named data object* specified in a *type declaration statement* containing the TARGET *attribute* or a *subobject* of such an object.

type (4). *Data type*.

type declaration statement (5). An INTEGER, REAL, DOUBLE PRECISION, COMPLEX, CHARACTER, LOGICAL, or TYPE(*type-name*) *statement*.

type parameter (2.4.1.1). A parameter of a parameterized *data type*.

type parameter values (2.4.1.1). The values of the *type parameters* of a *data entity* of parameterized *data type*.

undefined (2.5.4). For a *data object*, the property of not having a determinate value.

use association (14.6.1.2). The association of *names* in different *scoping units* specified by a USE *statement*.

variable (2.4.5). A *data object* whose value can be *defined* and redefined during the execution of an *executable program*. It may be a *named* data object, an *array element*, an *array section*, a *structure component*, or a *substring*. Note that in FORTRAN 77, a variable was always *scalar* and named.

vector subscript (6.2.2.5). A section subscript that is a rank-one integer expression.

whole array (6.2.1). A *named array*.

APPENDIX G. INDEX

(This appendix is not part of American National Standard X3.9-198x, but is included for information only.)

accessibility attribute 5-5
access-spec R510 5-5
access-stmt R521 5-9
ac-do-variable R434 4-10
ac-implied-do R432 4-10
ac-implied-do-control R433 4-10
action-stmt R215 2-2
action-term-do-construct R827 8-6
active 8-7
actual-arg R1216 12-6
actual-arg-spec R1214 12-6
ac-value R431 4-10
add-op R710 3-3
add-op R710 7-2
add-operand R706 7-2
advancing input/output statement 9-3
allocatable array 5-6
ALLOCATABLE attribute 5-8
ALLOCATE statement 6-6
allocate-stmt R619 6-6
allocation R621 6-6
alphanumeric-character R302 3-1
alt-return-spec R1217 12-7
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 8-12
array 2-7
array 6-2
array constructor 4-10
array element 2-7
array element order 6-3
array elements 6-2
array intrinsic assignment statement 7-18
array pointer 5-7
array section 2-7
array section 6-4
array-constructor R430 4-10
array-element R611 6-3
array-section R612 6-3
array-spec R512 5-5
ASCII collating sequence 3-2
assigned-goto-stmt R839 8-12
assignment-stmt R734 7-17
assign-stmt R838 8-12
association 2-9
assumed type parameter 5-2
assumed-shape array 5-6
assumed-shape-spec R516 5-6
assumed-size array 5-7
assumed-size-spec R518 5-7
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 5-17
blank-interp-edit-desc R1015 10-3
block 8-1
block R801 8-1
block-data R1110 11-5
block-data R1110 2-1
block-data-stmt R1111 11-5
block-do-construct R817 8-5
boz-literal-constant R407 4-3
branch target statement 8-11
Branching 8-11
c R1017 10-3
call-stmt R1212 12-6
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 7-8
character context 3-4
character intrinsic assignment statement 7-18
character intrinsic operation 7-4
character literal constant 4-5
character R301 3-1
character relational intrinsic operation 7-5
character set 3-1
character string 4-5
character string edit descriptor 10-2
character type 4-5
characteristics of a procedure 12-1
char-constant R309 3-3
char-constant-expr R730 7-8
char-expr R726 7-6
char-length R508 5-3
char-literal-constant R420 4-5
char-selector R506 5-3
char-string-edit-desc R1016 10-3
char-variable R604 6-1
CLOSE statement 9-8
close-spec R908 9-9
close-stmt R907 9-9
collating sequence 3-2
comment 3-5
comment 3-6
common block storage sequence 5-17
common blocks 5-17
COMMON statement 5-17

common-block-object R545 5-17
common-stmt R544 5-17
complex type 4-5
complex-literal-constant R417 4-5
component-array-spec R427 4-7
component-attr-stmt R426 4-7
component-decl R428 4-7
component-def-stmt R425 4-7
components 4-1
computed-goto-stmt R837 8-12
concatenation 4-6
concat-op R712 3-3
concat-op R712 7-3
conformable 2-7
connected 9-5
connect-spec R905 9-6
constant 2-7
constant expression 7-7
constant R305 3-3
constant-expr R729 7-8
constant-subobject R702 7-1
CONTAINS statement 12-12
contains-stmt R1229 12-12
continue-stmt R841 8-12
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 6-6
cycle-stmt R834 8-8
d R1008 10-2
data edit descriptor 10-2
data entity 2-6
data entity 4-2
data object 2-7
data object reference 2-9
DATA statement 5-10
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 R532 5-11
data-i-do-variable R533 5-11
data-implied-do R531 5-11
data-stmt R525 5-10
data-stmt-constant R529 5-11
data-stmt-object R527 5-11
data-stmt-repeat R530 5-11
data-stmt-set R526 5-11
data-stmt-value R528 5-11
DEALLOCATE statement 6-7
deallocate-stmt R623 6-7
declaration 2-8
declaration-construct R206 2-1
default character 4-5
default complex 4-5
default integer 4-3
default logical 4-6
default real 4-4
deferred-shape array 5-6
deferred-shape-spec R517 5-7
defined 2-8
defined assignment statement 7-18
defined binary operation 7-5
defined operation 7-5
defined unary operation 7-5
defined-assignment R1213 12-6
defined-binary-op R724 3-4
defined-binary-op R724 7-4
defined-operation R1210 12-6
defined-operator R311 3-4
defined-unary-op R704 3-4
defined-unary-op R704 7-2
definition 2-8
delete a file 9-2
deleted features 1-4
delimiter 3-4
derived type 2-6
derived-type intrinsic assignment statement 7-18
derived-type-def R422 4-7
derived-type-stmt R423 4-7
digits 3-1
digit-string R402 4-3
DIMENSION attribute 5-5
dimension-stmt R524 5-10
direct access input/output statement 9-11
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 8-5
do-term-action-stmt R829 8-6
do-term-shared-stmt R833 8-6
double precision real 4-4
do-variable R822 8-6
dummy procedure 12-1
dummy-arg R1225 12-11
e R1009 10-2
edit descriptor 10-2
element sequence 12-8
elemental 12-1
elemental function 13-1
elemental reference 12-9
else-if-stmt R804 8-2
else-stmt R805 8-2
elsewhere-stmt R741 7-21
END statement 2-5
end-block-data-stmt R1112 11-5
end-do R824 8-6
end-do-stmt R825 8-6
endfile record 9-1

endfile-stmt R920 9-18
end-function-stmt R1223 12-10
end-if-stmt R806 8-2
ending point 6-2
end-interface-stmt R1203 12-3
end-module-stmt R1106 11-2
end-of-file condition 9-14
end-program-stmt R1103 11-1
end-select-stmt R811 8-3
end-subroutine-stmt R1226 12-11
end-type-stmt R424 4-7
end-where-stmt R742 7-21
entity-decl R504 5-1
entry-stmt R1227 12-11
EQUIVALENCE statement 5-15
equivalence-object R543 5-15
equivalence-set R542 5-15
equivalence-stmt R541 5-15
equiv-op R722 3-4
equiv-op R722 7-4
equiv-operand R717 7-3
executable program 2-3
executable statement 2-4
executable-construct R214 2-2
executable-program R201 2-1
execution cycle 8-7
execution-part R207 2-2
execution-part-construct R208 2-2
exist 9-2
exit-stmt R835 8-8
explicit 12-2
explicit-shape array 5-6
explicit-shape-spec R513 5-6
exponent R415 4-4
exponent-letter R416 4-4
expr R723 7-4
expression 7-1
extension operation 7-5
extension operator 7-5
extent 2-7
EXTERNAL attribute 5-8
external file 9-2
external procedure 12-1
external procedure 2-4
EXTERNAL statement 12-5
external subprogram 2-3
external unit 9-5
external-file-unit R902 9-5
external-stmt R1207 12-5
external-subprogram R1217 12-10
external-subprogram R1217 2-1
external-subprogram R1218 12-9
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
free source form 3-4
function 2-3
function subprogram 12-9
function-reference R1209 12-6
function-stmt R1221 12-10
generic interface 12-3
Generic names 13-1
generic-spec R1206 12-3
global entity 14-1
goto-stmt R836 8-11
hex-constant R410 4-3
hex-digit R411 4-3
host 11-1
host 2-4
host association 11-2
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 12-2
IMPLICIT statement 5-13
implicit-part R204 2-1
implicit-part-stmt R205 2-1
implicit-spec R537 5-13
implicit-stmt R536 5-13
inactive 8-7
INCLUDE line 3-6
initial point 9-3
inner-shared-do-construct R832 8-6
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 9-19
inquire-stmt R923 9-19
inquiry function 13-1
instance 12-11
int-constant R308 3-3
int-constant-expr R731 7-8
integer constant expression 7-8
INTENT attribute 5-5
intent-spec R511 5-5

intent-stmt R519 5-9
interface 12-2
interface-block R1201 12-2
interface-stmt R1202 12-3
internal procedure 12-1
internal procedure 2-4
internal subprogram 2-3
internal unit 9-5
internal-file-unit R903 9-5
internal-subprogram R210 2-2
internal-subprogram-part R209 2-2
int-expr R727 7-6
int-literal-constant R404 4-3
intrinsic 2-9
intrinsic assignment statement 7-18
INTRINSIC attribute 5-8
intrinsic binary operation 7-4
intrinsic function 13-1
intrinsic operation 7-4
intrinsic procedure 12-1
INTRINSIC statement 12-6
intrinsic type 2-6
intrinsic unary operation 7-4
intrinsic-operator R310 3-3
intrinsic-stmt R1208 12-6
int-variable R605 6-1
io-control-spec R912 9-10
io-implied-do R916 9-13
io-implied-do-control R918 9-13
io-implied-do-object R917 9-13
io-unit R901 9-5
iteration count 8-7
k R1011 10-2
keyword 2-8
keyword R1215 12-6
kind 4-2
kind 4-4
kind 4-5
kind-param R405 4-3
kind-selector R505 5-1
label R313 3-4
label-do-stmt R819 8-6
length 4-5
length-selector R507 5-3
letters 3-1
letter-spec R538 5-13
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 9-11
literal constant 2-7
literal-constant R306 3-3
local entity 14-1
logical constant expression 7-8
logical intrinsic assignment statement 7-18
logical intrinsic operation 7-4
logical type 4-6
logical-constant-expr R732 7-8
logical-expr R725 7-6
logical-literal-constant R421 4-6
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 R1101 11-1
main-program R1101 2-1
many-one array section 6-5
masked array assignment 7-20
mask-expr R740 7-21
module 2-4
module procedure 12-1
module procedure 2-4
module R1104 11-2
module R1104 2-1
module reference 11-2
module subprogram 2-3
module-procedure-stmt R1205 12-3
module-stmt R1105 11-2
module-subprogram R212 2-2
module-subprogram-part R211 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 14-3
name R304 3-2
named common blocks 5-17
named constant 2-7
named file 9-2
named-constant R307 3-3
named-constant-def R535 5-12
namelist input/output statement 9-11
NAMELIST statement 5-14
namelist-group-object R540 5-14
namelist-stmt R539 5-14
Names 3-2
name-value subsequences 10-14
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
not-op R719 3-3
not-op R719 7-3 10 also
null value 9 13
NULLIFY statement 6-6
nullify-stmt R622 6-6

numeric constant expression 7-8
numeric intrinsic assignment statement 7-18
numeric intrinsic operation 7-4
numeric intrinsic operator 7-4
numeric relational intrinsic operation 7-5
numeric-expr R728 7-6
object 2-7
obsolescent features 1-4
octal-constant R409 4-3
only R1109 11-3
OPEN statement 9-6
open-stmt R904 9-6
operator 2-9
OPTIONAL attribute 5-8
optional-stmt R520 5-9
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
overloaded intrinsic operator 7-5
overloaded-intrinsic-op R312 3-4
overloads 12-4
PARAMETER attribute 5-4
PARAMETER statement 5-12
parameter-stmt R534 5-12
parent-array R613 6-3
parent-string R607 6-1
parent-structure R610 6-2
partially associated 14-5
pause-stmt R844 8-13
pointer 2-8
POINTER attribute 5-8
pointer-assignment-stmt R735 7-20
position 9-2
position-edit-desc R1012 10-2
position-spec R922 9-18
power-op R708 3-3
power-op R708 7-2
preceding record 9-3
Preconnection 9-6
prefix R1222 12-10
present 12-12
primary R701 7-1
PRINT statement 9-9
printing 9-17
print-stmt R911 9-9
PRIVATE 5-9
procedure 2-3
procedure interface 12-3
procedure interface block 2-4
procedure reference 2-9
procedure-ending R1219 12-11
procedure-ending R1220 12-10
procedure-ending R1220 2-1
procedure-heading R1218 12-11
procedure-heading R1218 2-1
procedure-heading R1219 12-10
procedure-interface R1204 12-3
processor 1-1
program name 11-1
program unit 2-3
program-stmt R1102 11-1
program-unit R202 2-1
PUBLIC 5-9
r R1004 10-2
range 8-7
range 8-7
rank 2-7
READ statement 9-9
reading 9-1
read-stmt R909 9-9
real part 4-5
real-literal-constant R413 4-4
real-part R418 4-5
record 9-1
record number 9-3
RECURSIVE 12-10
reference 6-1
relational intrinsic operation 7-4
rel-op R714 3-3
rel-op R714 7-3
rename R1108 11-3
repeat specification 10-2
representation methods 4-2
representation methods 4-5
restricted constant expression 7-7
restricted expression 7-8
RETURN statement 12-12
return-stmt R1228 12-12
rewind-stmt R921 9-18
SAVE attribute 5-8
saved object 5-8
saved-entity R523 5-10
save-stmt R522 5-10
scalar 2-7
scalar 6-1
scale factor 10-3
scope 14-1
scoping unit 2-3
section-subscript R615 6-3
select-case-stmt R809 8-3 — sequence?
sequential access input/output statement 9-11
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 2-7
shape conformance 7-7
share 8-7
shared-term-do-construct R831 8-6
sign R406 4-3
signed-digit-string R401 4-3

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 2-7
size of a common block 5-18
size of a storage sequence 14-4
source forms 3-4
special characters 3-1
specific interface 12-3
Specific names 13-1
specification expression 7-8
specification-expr R733 7-8
specification-part R203 2-1
specification-stmt R213 2-2
standard module 1-5
standard-conforming program 1-1
starting point 6-2
statement 3-4
statement entity 14-1
statement function 12-1
statement keyword 2-8
statement label 8-11
stat-variable R620 6-6
stmt-function-stmt R1230 12-14
stop-code R843 8-13
stop-stmt R842 8-13
storage associated 14-5
Storage association 14-4 4-7?
storage sequence 14-4
storage unit 14-4
storage units 2-8
stride 6-4
stride R617 6-3
structure 2-6
structure-component R609 6-2
structure-constructor R429 4-9
subobject designator 2-8
subobject R602 6-1
subroutine 2-3
subroutine subprogram 12-10
subroutine-reference R1211 12-6
subroutine-stmt R1224 12-11
subscript R614 6-3
subscript-triplet R616 6-3
substring 6-1
substring R606 6-1
substring-range R608 6-1
Syntax rules 1-2
TARGET attribute 5-8
target R736 7-20
terminal point 9-3
totally associated 14-5
transformational functions 13-1
type declaration statement 5-1
type specifier 5-3
type-declaration-stmt R501 5-1
type-param-value R509 5-3
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
upper-bound R515 5-6
use associated 11-3
Use association 14-3
USE statement 11-3
use-stmt R1107 11-3
value separator 10-12
variable 2-7
variable 6-1
variable R601 6-1
vector-subscript R618 6-3
w R1006 10-2
where-construct R738 7-20
where-construct-stmt R739 7-21
where-stmt R737 7-20
whole array 6-3
whole array named constant 6-3
whole array variable 6-3
WRITE statement 9-9
write-stmt R910 9-9
writing 9-1