
**Information technology — Programming
languages — Fortran —**

Part 1:
Base language

Technologies de l'information — Langages de programmation — Fortran —

Partie 1: Langage de base



Reference number
ISO/IEC 1539-1:1997

AVA Documents
Document Sharing Hub

Contents

	Foreword	xiii
	Introduction	xiv
1	Overview	1
1.1	Scope	1
1.2	Processor	1
1.3	Inclusions	1
1.4	Exclusions	1
1.5	Conformance	2
1.5.1	Fortran 90 compatibility	3
1.5.2	FORTTRAN 77 compatibility	3
1.6	Notation used in this standard	4
1.6.1	Informative notes	4
1.6.2	Syntax rules	4
1.6.3	Assumed syntax rules	5
1.6.4	Syntax conventions and characteristics	5
1.6.5	Text conventions	6
1.7	Deleted and obsolescent features	6
1.7.1	Nature of deleted features	6
1.7.2	Nature of obsolescent features	6
1.8	Modules	6
1.9	Normative references	7
2	Fortran terms and concepts	9
2.1	High level syntax	9
2.2	Program unit concepts	11
2.2.1	Program	12
2.2.2	Main program	12
2.2.3	Procedure	12
2.2.4	Module	13
2.3	Execution concepts	13
2.3.1	Executable/nonexecutable statements	13
2.3.2	Statement order	13
2.3.3	The END statement	14
2.3.4	Execution sequence	14

© ISO/IEC 1997

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

ISO/IEC Copyright Office • Case postale 56 • CH-1211 Genève 20 • Switzerland

Printed in Switzerland

2.4	Data concepts	15
2.4.1	Data type	15
2.4.2	Data value	15
2.4.3	Data entity	15
2.4.4	Scalar	17
2.4.5	Array	17
2.4.6	Pointer	17
2.4.7	Storage	17
2.5	Fundamental terms	17
2.5.1	Name and designator	18
2.5.2	Keyword	18
2.5.3	Declaration	18
2.5.4	Definition	18
2.5.5	Reference	18
2.5.6	Association	18
2.5.7	Intrinsic	19
2.5.8	Operator	19
2.5.9	Sequence	19
3	Characters, lexical tokens, and source form	21
3.1	Processor character set	21
3.1.1	Letters	21
3.1.2	Digits	21
3.1.3	Underscore	21
3.1.4	Special characters	22
3.1.5	Other characters	22
3.2	Low-level syntax	22
3.2.1	Names	22
3.2.2	Constants	23
3.2.3	Operators	23
3.2.4	Statement labels	24
3.2.5	Delimiters	24
3.3	Source form	24
3.3.1	Free source form	25
3.3.2	Fixed source form	27
3.4	Including source text	27
4	Intrinsic and derived data types	29
4.1	The concept of data type	29
4.1.1	Set of values	29
4.1.2	Constants	30
4.1.3	Operations	30
4.2	Relationship of types and values to objects	30
4.3	Intrinsic data types	31
4.3.1	Numeric types	31
4.3.2	Nonnumeric types	35
4.4	Derived types	37
4.4.1	Derived-type definition	38
4.4.2	Determination of derived types	43
4.4.3	Derived-type values	44
4.4.4	Construction of derived-type values	44
4.4.5	Derived-type operations and assignment	45

4.5	Construction of array values	45
5	Data object declarations and specifications	47
5.1	Type declaration statements	47
5.1.1	Type specifiers	50
5.1.2	Attributes	52
5.2	Attribute specification statements	57
5.2.1	INTENT statement	58
5.2.2	OPTIONAL statement	58
5.2.3	Accessibility statements	58
5.2.4	SAVE statement	59
5.2.5	DIMENSION statement	59
5.2.6	ALLOCATABLE statement	60
5.2.7	POINTER statement	60
5.2.8	TARGET statement	60
5.2.9	PARAMETER statement	60
5.2.10	DATA statement	61
5.3	IMPLICIT statement	63
5.4	NAMelist statement	65
5.5	Storage association of data objects	66
5.5.1	EQUIVALENCE statement	66
5.5.2	COMMON statement	68
6	Use of data objects	73
6.1	Scalars	74
6.1.1	Substrings	74
6.1.2	Structure components	75
6.2	Arrays	75
6.2.1	Whole arrays	76
6.2.2	Array elements and array sections	76
6.3	Dynamic association	79
6.3.1	ALLOCATE statement	79
6.3.2	NULLIFY statement	82
6.3.3	DEALLOCATE statement	82
7	Expressions and assignment	85
7.1	Expressions	85
7.1.1	Form of an expression	85
7.1.2	Intrinsic operations	89
7.1.3	Defined operations	90
7.1.4	Data type, type parameters, and shape of an expression	90
7.1.5	Conformability rules for elemental operations	92
7.1.6	Scalar and array expressions	93
7.1.7	Evaluation of operations	96
7.2	Interpretation of intrinsic operations	101
7.2.1	Numeric intrinsic operations	101
7.2.2	Character intrinsic operation	102
7.2.3	Relational intrinsic operations	102
7.2.4	Logical intrinsic operations	104
7.3	Interpretation of defined operations	104
7.3.1	Unary defined operation	104

7.3.2	Binary defined operation	105
7.4	Precedence of operators	105
7.5	Assignment	107
7.5.1	Assignment statement	107
7.5.2	Pointer assignment	110
7.5.3	Masked array assignment - WHERE	111
7.5.4	FORALL	114
8	Execution control	121
8.1	Executable constructs containing blocks	121
8.1.1	Rules governing blocks	121
8.1.2	IF construct	122
8.1.3	CASE construct	123
8.1.4	DO construct	126
8.2	Branching	130
8.2.1	Statement labels	130
8.2.2	GO TO statement	131
8.2.3	Computed GO TO statement	131
8.2.4	Arithmetic IF statement	131
8.3	CONTINUE statement	131
8.4	STOP statement	131
9	Input/output statements	133
9.1	Records	133
9.1.1	Formatted record	133
9.1.2	Unformatted record	133
9.1.3	Endfile record	134
9.2	Files	134
9.2.1	External files	134
9.2.2	Internal files	137
9.3	File connection	138
9.3.1	Unit existence	138
9.3.2	Connection of a file to a unit	138
9.3.3	Preconnection	139
9.3.4	The OPEN statement	139
9.3.5	The CLOSE statement	143
9.4	Data transfer statements	144
9.4.1	Control information list	144
9.4.2	Data transfer input/output list	148
9.4.3	Error, end-of-record, and end-of-file conditions	149
9.4.4	Execution of a data transfer input/output statement	150
9.4.5	Printing of formatted records	153
9.4.6	Termination of data transfer statements	154
9.5	File positioning statements	154
9.5.1	BACKSPACE statement	154
9.5.2	ENDFILE statement	155
9.5.3	REWIND statement	155
9.6	File inquiry	155
9.6.1	Inquiry specifiers	156
9.6.2	Restrictions on inquiry specifiers	160
9.6.3	Inquire by output list	160
9.7	Restrictions on function references and list items	160

9.8	Restriction on input/output statements	160
10	Input/output editing	161
10.1	Explicit format specification methods	161
10.1.1	FORMAT statement	161
10.1.2	Character format specification	161
10.2	Form of a format item list.	162
10.2.1	Edit descriptors	162
10.2.2	Fields	164
10.3	Interaction between input/output list and format	164
10.4	Positioning by format control	165
10.5	Data edit descriptors.	165
10.5.1	Numeric editing	165
10.5.2	Logical editing	170
10.5.3	Character editing	170
10.5.4	Generalized editing	170
10.6	Control edit descriptors	171
10.6.1	Position editing	172
10.6.2	Slash editing	173
10.6.3	Colon editing	173
10.6.4	S, SP, and SS editing.	173
10.6.5	P editing	173
10.6.6	BN and BZ editing	174
10.7	Character string edit descriptors.	174
10.8	List-directed formatting	174
10.8.1	List-directed input	175
10.8.2	List-directed output.	177
10.9	Namelist formatting	178
10.9.1	Namelist input	179
10.9.2	Namelist output	182
11	Program units	185
11.1	Main program	185
11.1.1	Main program specifications	185
11.1.2	Main program executable part.	186
11.1.3	Main program internal subprograms	186
11.2	External subprograms.	186
11.3	Modules	186
11.3.1	Module reference	187
11.3.2	The USE statement and use association	187
11.4	Block data program units.	189
12	Procedures	191
12.1	Procedure classifications	191
12.1.1	Procedure classification by reference.	191
12.1.2	Procedure classification by means of definition.	191
12.2	Characteristics of procedures.	192
12.2.1	Characteristics of dummy arguments	192
12.2.2	Characteristics of function results	192
12.3	Procedure interface	192
12.3.1	Implicit and explicit interfaces	192

12.3.2	Specification of the procedure interface	193
12.4	Procedure reference	198
12.4.1	Actual arguments, dummy arguments, and argument association	199
12.4.2	Function reference	205
12.4.3	Subroutine reference	206
12.5	Procedure definition	206
12.5.1	Intrinsic procedure definition	206
12.5.2	Procedures defined by subprograms	206
12.5.3	Definition of procedures by means other than Fortran	211
12.5.4	Statement function	211
12.6	Pure procedures	212
12.7	Elemental procedures	213
12.7.1	Elemental procedure declaration and interface	213
12.7.2	Elemental function actual arguments and results	214
12.7.3	Elemental subroutine actual arguments	214
13	Intrinsic procedures	217
13.1	Intrinsic functions	217
13.2	Elemental intrinsic procedures	217
13.3	Arguments to intrinsic procedures	217
13.4	Argument presence inquiry function	218
13.5	Numeric, mathematical, character, kind, logical, and bit procedures	218
13.5.1	Numeric functions	218
13.5.2	Mathematical functions	218
13.5.3	Character functions	218
13.5.4	Character inquiry function	218
13.5.5	Kind functions	218
13.5.6	Logical function	218
13.5.7	Bit manipulation and inquiry procedures	219
13.6	Transfer function	219
13.7	Numeric manipulation and inquiry functions	219
13.7.1	Models for integer and real data	219
13.7.2	Numeric inquiry functions	220
13.7.3	Floating point manipulation functions	220
13.8	Array intrinsic functions	220
13.8.1	The shape of array arguments	220
13.8.2	Mask arguments	221
13.8.3	Vector and matrix multiplication functions	221
13.8.4	Array reduction functions	221
13.8.5	Array inquiry functions	221
13.8.6	Array construction functions	221
13.8.7	Array reshape function	221
13.8.8	Array manipulation functions	222
13.8.9	Array location functions	222
13.9	Pointer association status functions	222
13.10	Intrinsic subroutines	222
13.10.1	Date and time subroutines	222
13.10.2	Pseudorandom numbers	222
13.10.3	Bit copy subroutine	222
13.11	Generic intrinsic functions	223
13.11.1	Argument presence inquiry function	223
13.11.2	Numeric functions	223
13.11.3	Mathematical functions	223

13.11.4	Character functions	224
13.11.5	Character inquiry function	224
13.11.6	Kind functions	224
13.11.7	Logical function	224
13.11.8	Numeric inquiry functions	224
13.11.9	Bit inquiry function	225
13.11.10	Bit manipulation functions	225
13.11.11	Transfer function	225
13.11.12	Floating-point manipulation functions	225
13.11.13	Vector and matrix multiply functions	225
13.11.14	Array reduction functions	225
13.11.15	Array inquiry functions	226
13.11.16	Array construction functions	226
13.11.17	Array reshape function	226
13.11.18	Array manipulation functions	226
13.11.19	Array location functions	226
13.11.20	Pointer association status functions	226
13.12	Intrinsic subroutines	226
13.13	Specific names for intrinsic functions	227
13.14	Specifications of the intrinsic procedures	228
14	Scope, association, and definition	275
14.1	Scope of names	275
14.1.1	Global entities	275
14.1.2	Local entities	275
14.1.3	Statement and construct entities	280
14.2	Scope of labels	281
14.3	Scope of external input/output units	281
14.4	Scope of operators	281
14.5	Scope of the assignment symbol	281
14.6	Association	281
14.6.1	Name association	282
14.6.2	Pointer association	284
14.6.3	Storage association	285
14.7	Definition and undefinition of variables	288
14.7.1	Definition of objects and subobjects	288
14.7.2	Variables that are always defined	288
14.7.3	Variables that are initially defined	288
14.7.4	Variables that are initially undefined	288
14.7.5	Events that cause variables to become defined	288
14.7.6	Events that cause variables to become undefined	290
A.	Glossary of technical terms	293
B.	Decremental features	303
B.1	Deleted features	303
B.1.1	Real and double precision DO variables	303
B.1.2	Branching to an END IF statement from outside its IF block	304
B.1.3	PAUSE statement	304
B.1.4	ASSIGN, assigned GO TO, and assigned FORMAT	304
B.1.5	H edit descriptor	306
B.2	Obsolescent features	306

B.2.1	Alternate return	306
B.2.2	Computed GO TO statement	307
B.2.3	Statement functions	307
B.2.4	DATA statements among executables	307
B.2.5	Assumed character length functions	307
B.2.6	Fixed form source	307
B.2.7	CHARACTER* form of CHARACTER declaration	307
C.	Extended notes	309
C.1	Section 4 notes	309
C.1.1	Intrinsic and derived data types (4.3, 4.4)	309
C.1.2	Selection of the approximation methods (4.3.1.2)	310
C.1.3	Pointers (4.4.1)	311
C.2	Section 5 notes	312
C.2.1	The POINTER attribute (5.1.2.7)	312
C.2.2	The TARGET attribute (5.1.2.8)	312
C.3	Section 6 notes	313
C.3.1	Structure components (6.1.2)	313
C.3.2	Pointer allocation and association	314
C.4	Section 7 notes	314
C.4.1	Character assignment	314
C.4.2	Evaluation of function references	315
C.4.3	Pointers in expressions	315
C.4.4	Pointers on the left side of an assignment	315
C.4.5	An example of a FORALL construct containing a WHERE construct	316
C.4.6	Examples of FORALL statements	316
C.5	Section 8 notes	317
C.5.1	Loop control	317
C.5.2	The CASE construct	317
C.5.3	Additional examples of DO constructs	317
C.5.4	Examples of invalid DO constructs	319
C.6	Section 9 notes	319
C.6.1	Files (9.2)	319
C.6.2	OPEN statement (9.3.4)	322
C.6.3	Connection properties (9.3.2)	323
C.6.4	CLOSE statement (9.3.5)	324
C.6.5	INQUIRE statement (9.6)	325
C.7	Section 10 notes	325
C.7.1	Number of records (10.3, 10.4, 10.6.2)	325
C.7.2	List-directed input (10.8.1)	326
C.8	Section 11 notes	327
C.8.1	Main program and block data program unit (11.1, 11.4)	327
C.8.2	Dependent compilation (11.3)	327
C.8.3	Examples of the use of modules	329
C.9	Section 12 notes	334
C.9.1	Portability problems with external procedures (12.3.2.2)	334
C.9.2	Procedures defined by means other than Fortran (12.5.3)	334
C.9.3	Procedure interfaces (12.3)	335
C.9.4	Argument association and evaluation (12.4.1.1)	335
C.9.5	Pointers and targets as arguments (12.4.1.1)	336
C.10	Section 14 notes	337
C.10.1	Examples of host association (14.6.1.3)	337
C.11	Array feature notes	338

C.11.1	Summary of features	338
C.11.2	Examples	339
C.11.3	FORmula TRANslation and array processing	343
C.11.4	Sum of squared residuals	344
C.11.5	Vector norms: infinity-norm and one-norm	344
C.11.6	Matrix norms: infinity-norm and one-norm	344
C.11.7	Logical queries	344
C.11.8	Parallel computations	345
C.11.9	Example of element-by-element computation	345
C.11.10	Bit manipulation and inquiry procedures	346
D.	Index	347

Tables

Table 2.1	Requirements on statement ordering	13
Table 2.2	Statements allowed in scoping units	14
Table 3.1	Special characters	22
Table 6.1	Subscript order value	77
Table 7.1	Type of operands and results for intrinsic operators	89
Table 7.2	Type, type parameters, and rank of the result of NULL ()	91
Table 7.3	Interpretation of the numeric intrinsic operators	101
Table 7.4	Interpretation of the character intrinsic operator //	102
Table 7.5	Interpretation of the relational intrinsic operators	103
Table 7.6	Interpretation of the logical intrinsic operators	104
Table 7.7	The values of operations involving logical intrinsic operators	104
Table 7.8	Categories of operations and relative precedence	105
Table 7.9	Type conformance for the intrinsic assignment statement	108
Table 7.10	Numeric conversion and the assignment statement	108
Table C.1	Values assigned to INQUIRE specifier variables	325

Foreword

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

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

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

This edition cancels and replaces ISO/IEC 1539:1991, which has been technically revised.

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

- *Part 1: Base language*
- *Part 2: Varying length character strings*

Annexes A to D of this part of ISO/IEC 1539 are for information only.

Introduction

Standard programming language Fortran

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

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

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

- (1) The FORALL statement and construct
- (2) PURE and ELEMENTAL procedures
- (3) Pointer initialization and structure default initialization

FORALL

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

PURE

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

Initialization

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

Organization of this part of ISO/IEC 1539

This part of ISO/IEC 1539 is organized in 14 sections, dealing with 7 conceptual areas. These 7 areas, and the sections in which they are treated, are :

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

High/low level concepts

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

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

Data concepts

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

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

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

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

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

Computations

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

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

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

Execution control

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

Input/output

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

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

Program units

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

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

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

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

Scoping and association rules

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

Annexes

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

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

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

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

Information technology — Programming languages — Fortran —

Part 1: Base language

Section 1: Overview

1.1 Scope

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

1.2 Processor

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

1.3 Inclusions

This standard specifies

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

1.4 Exclusions

This standard does not specify

- (1) The mechanism by which programs are transformed for use on computing systems,
- (2) The operations required for setup and control of the use of programs on computing systems,
- (3) The method of transcription of programs or their input or output data to or from a storage medium,
- (4) The program and processor behavior when this standard fails to establish an interpretation except for the processor detection and reporting requirements in items (2) through (8) of 1.5,
- (5) The size or complexity of a program and its data that will exceed the capacity of any specific computing system or the capability of a particular processor,
- (6) The physical properties of the representation of quantities and the method of rounding, approximating, or computing numeric values on a particular processor,

- (7) The physical properties of input/output records, files, and units, or
- (8) The physical properties and implementation of storage.

1.5 Conformance

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

A processor conforms to this standard if

- (1) It executes any standard-conforming program in a manner that fulfills the interpretations herein, subject to any limits that the processor may impose on the size and complexity of the program;
- (2) It contains the capability to detect and report the use within a submitted program unit of a form designated herein as obsolescent, insofar as such use can be detected by reference to the numbered syntax rules and their associated constraints;
- (3) It contains the capability to detect and report the use within a submitted program unit of an additional form or relationship that is not permitted by the numbered syntax rules or their associated constraints, including the deleted features described in Annex B;
- (4) It contains the capability to detect and report the use within a submitted program unit of kind type parameter values (4.3) not supported by the processor;
- (5) It contains the capability to detect and report the use within a submitted program unit of source form or characters not permitted by Section 3;
- (6) It contains the capability to detect and report the use within a submitted program of name usage not consistent with the scope rules for names, labels, operators, and assignment symbols in Section 14;
- (7) It contains the capability to detect and report the use within a submitted program unit of intrinsic procedures whose names are not defined in Section 13; and
- (8) It contains the capability to detect and report the reason for rejecting a submitted program.

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

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

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

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

NOTE 1.1

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

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

1.5.1 Fortran 90 compatibility

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

- (1) Real and double precision DO variables,
- (2) Branching to an ENDIF statement from outside its IF construct,
- (3) PAUSE statement,
- (4) ASSIGN and assigned GOTO statements and assigned format specifiers, and
- (5) H edit descriptor.

NOTE 1.2

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

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

- (1) If the processor can distinguish between positive and negative real zero, the behavior of the SIGN intrinsic function when the second argument is negative real zero is changed by this standard.
- (2) This standard has more intrinsic procedures than did Fortran 90. Therefore, a standard-conforming Fortran 90 program may have a different interpretation under this standard if it invokes an external procedure having the same name as one of the new standard intrinsic procedures, unless that procedure is specified in an EXTERNAL statement or an interface body.

1.5.2 FORTRAN 77 compatibility

Except as noted in this section, the Fortran 95 Standard is an upward compatible extension to the earlier Fortran International Standard, ISO 1539:1980, informally referred to as FORTRAN 77. Any standard-conforming FORTRAN 77 program that does not use one of the deleted features listed in 1.5.1 remains standard conforming under the Fortran 95 Standard; however, see item (4) below regarding intrinsic procedures. The Fortran 95 Standard restricts the behavior for some features that were processor dependent in FORTRAN 77. Therefore, a standard-conforming FORTRAN 77 program that uses one of these processor-dependent features may have a different interpretation under the Fortran 95 Standard, yet remain a standard-conforming program. The following FORTRAN 77 features have different interpretations in the Fortran 95 Standard:

- (1) FORTRAN 77 permitted a processor to supply more precision derived from a real constant than can be represented in a real datum when the constant is used to initialize a data object of type double precision real in a DATA statement. The Fortran 95 Standard does not permit a processor this option.

- (2) If a named variable that was not in a common block was initialized in a DATA statement and did not have the SAVE attribute specified, FORTRAN 77 left its SAVE attribute processor dependent. The Fortran 95 Standard specifies (5.2.10) that this named variable has the SAVE attribute.
- (3) FORTRAN 77 required that the number of characters required by the input list was to be less than or equal to the number of characters in the record during formatted input. The Fortran 95 Standard specifies (9.4.4.4.2) that the input record is logically padded with blanks if there are not enough characters in the record, unless the PAD= specifier with the value 'NO' is specified in an appropriate OPEN statement.
- (4) The Fortran 95 Standard has more intrinsic functions than did FORTRAN 77 and adds a few intrinsic subroutines. Therefore, a standard-conforming FORTRAN 77 program may have a different interpretation under the Fortran 95 Standard if it invokes an external procedure having the same name as one of the new standard intrinsic procedures, unless that procedure is specified in an EXTERNAL statement as recommended for nonintrinsic functions in appendix B of the FORTRAN 77 standard.
- (5) A value of 0 for a list item in a formatted output statement will be formatted in a different form for some G edit descriptors. In addition, the Fortran 95 standard specifies how rounding of values will affect the output field form, but FORTRAN 77 did not address this issue: therefore, some FORTRAN 77 processors may produce a different output form than Fortran 95 processors for certain combinations of values and G edit descriptors.
- (6) If the processor can distinguish between positive and negative real zero, the behavior of the SIGN intrinsic function when the second argument is negative real zero is changed by this standard.

1.6 Notation used in this standard

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

1.6.1 Informative notes

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

1.6.2 Syntax rules

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

- (1) Characters from the Fortran character set (3.1) are interpreted literally as shown, except where otherwise noted.
- (2) Lower-case italicized letters and words (often hyphenated and abbreviated) represent general syntactic classes for which specific syntactic entities shall be substituted in actual statements.

Common abbreviations used in syntactic terms are:

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

- (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 two-digit sequence number within that section. The syntax rules are distributed as appropriate throughout the text, and are referenced by number as needed. Some rules in Sections 2 and 3 are more fully described in later sections; in such cases, the section number s is the number of the later section where the rule is repeated.
- (5) The syntax rules are not a complete and accurate syntax description of Fortran, and cannot be used to automatically generate a Fortran parser; where a syntax rule is incomplete, it is restricted by the corresponding constraints and text.

NOTE 1.3

An example of the use of the syntax rules is:

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

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

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

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

4
67
1999
10243852

1.6.3 Assumed syntax rules

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

<i>xyz-list</i>	is <i>xyz</i> [, <i>xyz</i>] ...
<i>xyz-name</i>	is <i>name</i>
<i>scalar-xyz</i>	is <i>xyz</i>

Constraint: *scalar-xyz* shall be scalar.

1.6.4 Syntax conventions and characteristics

- (1) Any syntactic class name ending in "-stmt" follows the source form statement rules: it shall be delimited by end-of-line or semicolon, and may be labeled unless it forms part of another statement (such as an IF or WHERE statement). Conversely, everything considered to be a source form statement is given a "-stmt" ending in the syntax rules.
- (2) The rules on statement ordering are described rigorously in the definition of *program-unit* (R202). Expression hierarchy is described rigorously in the definition of *expr* (R723).

- (3) The suffix *"-spec"* is used consistently for specifiers, such as input/output statement specifiers. It also is used for type declaration attribute specifications (for example, *"array-spec"* in R513), and in a few other cases.
- (4) When reference is made to a type parameter, including the surrounding parentheses, the suffix *"-selector"* is used. See, for example, *"kind-selector"* (R506) and *"length-selector"* (R508).
- (5) The term *"subscript"* (for example, R617, R618, and R619) is used consistently in array definitions.

1.6.5 Text conventions

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

NOTE 1.4

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

1.7 Deleted and obsolescent features

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

1.7.1 Nature of deleted features

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

1.7.2 Nature of obsolescent features

- (1) Better methods existed in Fortran 90.
- (2) It is recommended that programmers should use these better methods in new programs and convert existing code to these methods.
- (3) These features are identified in the text of this document by a distinguishing type font (1.6.5).
- (4) If the use of these features has become insignificant in Fortran programs, future Fortran standards committees should consider deleting them from the next revision.
- (5) The next Fortran standards committee should consider for deletion only those language features that appear in the list of obsolescent features.
- (6) Processors supporting the Fortran language should support these features as long as they continue to be used widely in Fortran programs.

1.8 Modules

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

1.9 Normative references

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

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

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

Section 2: Fortran terms and concepts

2.1 High level syntax

This section introduces the terms associated with program units and other Fortran concepts above the construct, statement, and expression levels and illustrates their relationships. The notation used in this standard is described in 1.6.

NOTE 2.1

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

R201	<i>program</i>	is <i>program-unit</i> [<i>program-unit</i>] ...
A <i>program</i> shall contain exactly one <i>main-program program-unit</i> .		
R202	<i>program-unit</i>	is <i>main-program</i> or <i>external-subprogram</i> or <i>module</i> or <i>block-data</i>
R1101	<i>main-program</i>	is [<i>program-stmt</i>] [<i>specification-part</i>] [<i>execution-part</i>] [<i>internal-subprogram-part</i>] <i>end-program-stmt</i>
R203	<i>external-subprogram</i>	is <i>function-subprogram</i> or <i>subroutine-subprogram</i>
R1216	<i>function-subprogram</i>	is <i>function-stmt</i> [<i>specification-part</i>] [<i>execution-part</i>] [<i>internal-subprogram-part</i>] <i>end-function-stmt</i>
R1221	<i>subroutine-subprogram</i>	is <i>subroutine-stmt</i> [<i>specification-part</i>] [<i>execution-part</i>] [<i>internal-subprogram-part</i>] <i>end-subroutine-stmt</i>
R1104	<i>module</i>	is <i>module-stmt</i> [<i>specification-part</i>] [<i>module-subprogram-part</i>] <i>end-module-stmt</i>
R1112	<i>block-data</i>	is <i>block-data-stmt</i> [<i>specification-part</i>] <i>end-block-data-stmt</i>
R204	<i>specification-part</i>	is [<i>use-stmt</i>] ... [<i>implicit-part</i>] [<i>declaration-construct</i>] ...

R205	<i>implicit-part</i>	is [<i>implicit-part-stmt</i>] ... <i>implicit-stmt</i>
R206	<i>implicit-part-stmt</i>	is <i>implicit-stmt</i> or <i>parameter-stmt</i> or <i>format-stmt</i> or <i>entry-stmt</i>
R207	<i>declaration-construct</i>	is <i>derived-type-def</i> or <i>interface-block</i> or <i>type-declaration-stmt</i> or <i>specification-stmt</i> or <i>parameter-stmt</i> or <i>format-stmt</i> or <i>entry-stmt</i> or <i>stmt-function-stmt</i>
R208	<i>execution-part</i>	is <i>executable-construct</i> [<i>execution-part-construct</i>] ...
R209	<i>execution-part-construct</i>	is <i>executable-construct</i> or <i>format-stmt</i> or <i>entry-stmt</i> or <i>data-stmt</i>
R210	<i>internal-subprogram-part</i>	is <i>contains-stmt</i> <i>internal-subprogram</i> [<i>internal-subprogram</i>] ...
R211	<i>internal-subprogram</i>	is <i>function-subprogram</i> or <i>subroutine-subprogram</i>
R212	<i>module-subprogram-part</i>	is <i>contains-stmt</i> <i>module-subprogram</i> [<i>module-subprogram</i>] ...
R213	<i>module-subprogram</i>	is <i>function-subprogram</i> or <i>subroutine-subprogram</i>
R214	<i>specification-stmt</i>	is <i>access-stmt</i> or <i>allocatable-stmt</i> or <i>common-stmt</i> or <i>data-stmt</i> or <i>dimension-stmt</i> or <i>equivalence-stmt</i> or <i>external-stmt</i> or <i>intent-stmt</i> or <i>intrinsic-stmt</i> or <i>namelist-stmt</i> or <i>optional-stmt</i> or <i>pointer-stmt</i> or <i>save-stmt</i> or <i>target-stmt</i>
R215	<i>executable-construct</i>	is <i>action-stmt</i> or <i>case-construct</i> or <i>do-construct</i> or <i>forall-construct</i> or <i>if-construct</i>

	<i>or where-construct</i>
R216 <i>action-stmt</i>	<i>is allocate-stmt</i>
	<i>or assignment-stmt</i>
	<i>or backspace-stmt</i>
	<i>or call-stmt</i>
	<i>or close-stmt</i>
	<i>or continue-stmt</i>
	<i>or cycle-stmt</i>
	<i>or deallocate-stmt</i>
	<i>or endfile-stmt</i>
	<i>or end-function-stmt</i>
	<i>or end-program-stmt</i>
	<i>or end-subroutine-stmt</i>
	<i>or exit-stmt</i>
	<i>or forall-stmt</i>
	<i>or goto-stmt</i>
	<i>or if-stmt</i>
	<i>or inquire-stmt</i>
	<i>or nullify-stmt</i>
	<i>or open-stmt</i>
	<i>or pointer-assignment-stmt</i>
	<i>or print-stmt</i>
	<i>or read-stmt</i>
	<i>or return-stmt</i>
	<i>or rewind-stmt</i>
	<i>or stop-stmt</i>
	<i>or where-stmt</i>
	<i>or write-stmt</i>
	<i>or arithmetic-if-stmt</i>
	<i>or computed-goto-stmt</i>

Constraint: An *execution-part* shall not contain an *end-function-stmt*, *end-program-stmt*, or *end-subroutine-stmt*.

2.2 Program unit concepts

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

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

- (1) A derived-type definition (4.4.1),
- (2) A procedure interface body, excluding any derived-type definitions and procedure interface bodies in it (12.3.2.1), or
- (3) A program unit or subprogram, excluding derived-type definitions, procedure interface bodies, and subprograms in it.

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

2.2.1 Program

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

NOTE 2.2

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

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

2.2.2 Main program

The main program is described in 11.1.

2.2.3 Procedure

A **procedure** encapsulates an arbitrary sequence of computations that may be invoked directly during program execution. Procedures are either functions or subroutines. A **function** is a procedure that is invoked in an expression; its invocation causes a value to be computed which is then used in evaluating the expression. The variable that returns the value of a function is called the **result variable**. A **subroutine** is a procedure that is invoked in a CALL statement or by a defined assignment statement (12.4, 12.4.3, 7.5.1.3). Unless it is a pure procedure, a subroutine may be used to change the program state by changing the values of any of the data objects accessible to the subroutine; unless it is a pure procedure, a function may do this in addition to computing the function value.

Procedures are described further in Section 12.

2.2.3.1 External procedure

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

2.2.3.2 Module procedure

A **module procedure** is a procedure that is defined by a module subprogram (R213). A module procedure may be invoked by another module subprogram in the module or by any scoping unit that accesses the module procedure by use association (11.3.2). The module containing the subprogram is called the **host** of the module procedure.

2.2.3.3 Internal procedure

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

2.2.3.4 Procedure interface block

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

2.2.4 Module

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

2.3 Execution concepts

Each Fortran statement is classified as either an executable statement or a nonexecutable statement. There are restrictions on the order in which statements may appear in a program unit, and certain executable statements may appear only in certain executable constructs.

2.3.1 Executable/nonexecutable statements

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

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

Table 2.1 Requirements on statement ordering

PROGRAM, FUNCTION, SUBROUTINE, MODULE, or BLOCK DATA statement		
USE statements		
FORMAT and ENTRY statements	IMPLICIT NONE	
	PARAMETER statements	IMPLICIT statements
	PARAMETER and DATA statements	Derived-type definitions, interface blocks, type declaration statements, specification statements, and statement function statements
	DATA statements	Executable constructs
CONTAINS statement		
Internal subprograms or module subprograms		
END statement		

2.3.2 Statement order

The syntax rules of subclause 2.1 specify the statement order within program units and subprograms. These rules are illustrated in Table 2.1 and Table 2.2. Table 2.1 shows the ordering rules for statements and applies to all program units and subprograms. Vertical lines delineate varieties of statements that may be interspersed and horizontal lines delineate varieties of statements that shall not be interspersed. Internal or module subprograms shall follow a

CONTAINS statement. Between USE and CONTAINS statements in a subprogram, nonexecutable statements generally precede executable statements, although the ENTRY statement, FORMAT statement, and DATA statement may appear among the executable statements. Table 2.2 shows which statements are allowed in a scoping unit.

Table 2.2 Statements allowed in scoping units

Kind of scoping unit:	Main program	Module	Block data	External subprog	Module subprog	Internal subprog	Interface body
USE statement	Yes	Yes	Yes	Yes	Yes	Yes	Yes
ENTRY statement	No	No	No	Yes	Yes	No	No
FORMAT statement	Yes	No	No	Yes	Yes	Yes	No
Misc. declarations (see note)	Yes	Yes	Yes	Yes	Yes	Yes	Yes
DATA statement	Yes	Yes	Yes	Yes	Yes	Yes	No
Derived-type definition	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Interface block	Yes	Yes	No	Yes	Yes	Yes	Yes
Executable statement	Yes	No	No	Yes	Yes	Yes	No
CONTAINS statement	Yes	Yes	No	Yes	Yes	No	No
Statement function statement	Yes	No	No	Yes	Yes	Yes	No
Notes for Table 2.2: 1) Misc. declarations are PARAMETER statements, IMPLICIT statements, type declaration statements, and specification statements. 2) Derived type definitions are also scoping units, but they do not contain any of the above statements, and so have not been listed in the table. 3) The scoping unit of a module does not include any module subprograms that the module contains.							

2.3.3 The END statement

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

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

2.3.4 Execution sequence

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

- (1) Execution of a branching statement (8.2) changes the execution sequence. These statements explicitly specify a new starting place for the execution sequence.
- (2) CASE constructs, DO constructs, and IF constructs contain an internal statement structure and execution of these constructs involves implicit internal branching. See Section 8 for the detailed semantics of each of these constructs.
- (3) END=, ERR=, and EOR= specifiers may result in a branch.
- (4) Alternate returns may result in a branch.

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

2.4 Data concepts

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

2.4.1 Data type

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

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

2.4.1.1 Intrinsic type

An **intrinsic type** is a type that is defined implicitly, along with operations, and is always accessible. The intrinsic types are integer, real, complex, character, and logical. The properties of intrinsic types are described in 4.3. An intrinsic type may be parameterized, in which case the set of data values depends on the values of the parameters. Such a parameter is called a **type parameter** (4.3). The type parameters are KIND and LEN.

The **kind type parameter** indicates the decimal exponent range for the integer type (4.3.1.1), the decimal precision and exponent range for the real and complex types (4.3.1.2, 4.3.1.3), and the representation methods for the character and logical types (4.3.2.1, 4.3.2.2). The **character length parameter** specifies the number of characters for the character type.

2.4.1.2 Derived type

A **derived type** is a type that is not defined implicitly but requires a type definition to declare components of intrinsic or of other derived types. A scalar object of such a derived type is called a **structure** (5.1.1.7). The only intrinsic operation for derived types is assignment with type agreement (7.5.1.5). For each derived type, structure constructors are available to provide values (4.4.4). In addition, data objects of derived type may be used as procedure arguments and function results, and may appear in input/output lists. If additional operations are needed for a derived type, they shall be supplied as procedure definitions.

Derived types are described further in 4.4.

2.4.2 Data value

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

2.4.3 Data entity

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

2.4.3.1 Data object

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

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

Objects referenced by a name are:

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

Subobjects referenced by a subobject designator are:

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

2.4.3.1.1 Variable

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

2.4.3.1.2 Constant

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

2.4.3.1.3 Subobject of a constant

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

NOTE 2.3

For example, given:

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

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

2.4.3.2 Expression

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

2.4.3.3 Function reference

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

2.4.4 Scalar

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

NOTE 2.4

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

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

2.4.5 Array

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

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

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

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

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

2.4.6 Pointer

A **pointer** is a variable that has the POINTER attribute. A pointer is **associated** with a target by allocation (6.3.1) or pointer assignment (7.5.2). A pointer shall neither be referenced nor defined until it is associated. A pointer is **disassociated** following execution of a DEALLOCATE or NULLIFY statement, following pointer association with a disassociated pointer, or initially through pointer initialization. A disassociated pointer is not currently associated with a target (14.6.2). If the pointer is an array, the rank is declared, but the extents are determined when the pointer is associated with a target.

2.4.7 Storage

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

2.5 Fundamental terms

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

2.5.1 Name and designator

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

2.5.2 Keyword

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

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

NOTE 2.5

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

2.5.3 Declaration

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

2.5.4 Definition

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

2.5.5 Reference

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

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

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

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

2.5.6 Association

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

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

2.5.7 Intrinsic

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

2.5.8 Operator

An **operator** specifies a particular computation involving one (unary operator) or two (binary operator) data values (**operands**). Fortran contains a number of intrinsic operators (e.g., the arithmetic operators $+$, $-$, $*$, $/$, and $**$ with numeric operands and the logical operators **.AND.**, **.OR.**, etc. with logical operands). Additional operators may be defined within a program (7.1.3).

2.5.9 Sequence

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

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

Section 3: Characters, lexical tokens, and source form

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

3.1 Processor character set

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

- (1) **Control characters** ("newline", for example)
- (2) **Graphic characters**
 - (a) Letters (3.1.1)
 - (b) Digits (3.1.2)
 - (c) Underscore (3.1.3)
 - (d) Special characters (3.1.4)
 - (e) Other characters (3.1.5)

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

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

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

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

3.1.1 Letters

The twenty-six **letters** are:

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

The set of letters defines the syntactic class *letter*.

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

3.1.2 Digits

The ten **digits** are:

0 1 2 3 4 5 6 7 8 9

The ten digits define the syntactic class *digit*.

3.1.3 Underscore

R303 *underscore* **is** *_*

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

3.1.4 Special characters

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

Table 3.1 Special characters

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

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

3.1.5 Other characters

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

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

3.2 Low-level syntax

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

3.2.1 Names

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

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

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

NOTE 3.1

Examples of names:

A1**NAME_LENGTH**

(single underscore)

S_P_R_E_A_D_O_U_T

(two consecutive underscores)

TRAILER_

(trailing underscore)

3.2.2 Constants

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

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

R307 *named-constant* is *name*

R308 *int-constant* is *constant*

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

R309 *char-constant* is *constant*

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

3.2.3 Operators

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

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

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

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

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

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

		or <=
		or >
		or >=
R719	<i>not-op</i>	is .NOT.
R720	<i>and-op</i>	is .AND.
R721	<i>or-op</i>	is .OR.
R722	<i>equiv-op</i>	is .EQV.
		or .NEQV.
R311	<i>defined-operator</i>	is <i>defined-unary-op</i>
		or <i>defined-binary-op</i>
		or <i>extended-intrinsic-op</i>
R704	<i>defined-unary-op</i>	is . letter [letter]
R724	<i>defined-binary-op</i>	is . letter [letter]
R312	<i>extended-intrinsic-op</i>	is <i>intrinsic-operator</i>
Constraint: A <i>defined-unary-op</i> and a <i>defined-binary-op</i> shall not contain more than 31 letters and shall not be the same as any <i>intrinsic-operator</i> or <i>logical-literal-constant</i> .		

3.2.4 Statement labels

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

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

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

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

NOTE 3.2

For example:

99999

10

010

are all statement labels. The last two are equivalent.

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

3.2.5 Delimiters

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

```
( ... )
/ ... /
(/ ... /)
```

3.3 Source form

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

A **character context** means characters within a character literal constant (4.3.2.1) or within a character string edit descriptor (10.2.1).

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

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

3.3.1 Free source form

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

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

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

NOTE 3.3

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

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

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

Adjacent keywords where separating
blanks are optional

BLOCK DATA
DOUBLE PRECISION
ELSE IF
END BLOCK DATA
END DO
END FILE
END FORALL
END FUNCTION
END IF
END INTERFACE
END MODULE
END PROGRAM
END SELECT
END SUBROUTINE
END TYPE
END WHERE
GO TO
IN OUT
SELECT CASE

NOTE 3.4

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

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

- (1) Confine statement labels to character positions 1 to 5 and statements to character positions 7 to 72;
- (2) Treat blanks as being significant;
- (3) Use only the exclamation mark (!) to indicate a comment, but do not start the comment in character position 6;
- (4) For continued statements, place an ampersand (&) in both character position 73 of a continued line and character position 6 of a continuing line.

3.3.1.1 Free form commentary

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

NOTE 3.5

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

3.3.1.2 Free form statement separation

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

3.3.1.3 Free form statement continuation

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

3.3.1.3.1 Noncharacter context continuation

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

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

3.3.1.3.2 Character context continuation

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

3.3.1.4 Free form statements

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

NOTE 3.6

No Fortran statement begins with a digit.

A free form statement shall not have more than 39 continuation lines.

3.3.2 Fixed source form

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

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

3.3.2.1 Fixed form commentary

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

NOTE 3.7

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

3.3.2.2 Fixed form statement separation

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

3.3.2.3 Fixed form statement continuation

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

NOTE 3.8

An "!" or ";" in character position 6 is interpreted as a continuation indicator unless it appears within commentary indicated by a "C" or "*" in character position 1 or by an "!" in character positions 1–5 (3.3.2.3).

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

3.3.2.4 Fixed form statements

A label, if present, shall occur in character positions 1 through 5 of the first line of a statement; otherwise, positions 1 through 5 shall be blank. Blanks may appear anywhere within a label. A statement following a ";" on the same line shall not be labeled. Character positions 1 through 5 of any continuation lines shall be blank. A fixed form statement shall not have more than 19 continuation lines. The program unit END statement shall not be continued. A statement whose initial line appears to be a program unit END statement shall not be continued.

3.4 Including source text

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

INCLUDE *char-literal-constant*

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

An INCLUDE line is not a Fortran statement.

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

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

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

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

Section 4: Intrinsic and derived data types

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

NOTE 4.1

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

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

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

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

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

NOTE 4.2

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

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

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

4.1 The concept of data type

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

4.1.1 Set of values

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

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

4.1.2 Constants

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

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

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

4.1.3 Operations

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

For derived types, the only intrinsic operation is assignment. All other operations shall be defined by the program (4.4.5).

4.2 Relationship of types and values to objects

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

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

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

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

NOTE 4.3

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

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

4.3 Intrinsic data types

The intrinsic data types are:

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

NOTE 4.4

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

4.3.1 Numeric types

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

4.3.1.1 Integer type

The set of values for the **integer type** is a subset of the mathematical integers. A processor shall provide one or more **representation methods** that define sets of values for data of type integer. Each such method is characterized by a value for a type parameter called the **kind type parameter**. The kind type parameter of a representation method is returned by the intrinsic inquiry function **KIND** (13.14.52). The decimal exponent range of a representation method is returned by the intrinsic function **RANGE** (13.14.87). The intrinsic function **SELECTED_INT_KIND** (13.14.94) returns a kind value based on a specified decimal range requirement. The integer type includes a zero value, which is considered neither negative nor positive. The value of a signed integer zero is the same as the value of an unsigned integer zero.

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

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

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

R401	<i>signed-digit-string</i>	is	[<i>sign</i>] <i>digit-string</i>
R402	<i>digit-string</i>	is	<i>digit</i> [<i>digit</i>] ...
R403	<i>signed-int-literal-constant</i>	is	[<i>sign</i>] <i>int-literal-constant</i>
R404	<i>int-literal-constant</i>	is	<i>digit-string</i> [<i>_ kind-param</i>]
R405	<i>kind-param</i>	is	<i>digit-string</i> or <i>scalar-int-constant-name</i>
R406	<i>sign</i>	is	+ or -

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

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

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

An integer constant is interpreted as a decimal value.

NOTE 4.5

Examples of signed integer literal constants are:

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

where SHORT is a scalar integer named constant.

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

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

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

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

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

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

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

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

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

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

4.3.1.2 Real type

The **real type** has values that approximate the mathematical real numbers. A processor shall provide two or more **approximation methods** that define sets of values for data of type real. Each such method has a **representation method** and is characterized by a value for a type parameter called the **kind** type parameter. The kind type parameter of an approximation method is returned by the intrinsic inquiry function KIND (13.14.52). The decimal precision and decimal exponent range of an approximation method are returned by the intrinsic functions PRECISION (13.14.81) and RANGE (13.14.87). The intrinsic function SELECTED_REAL_KIND (13.14.95) returns a kind value based on specified precision and decimal range requirements.

NOTE 4.6

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